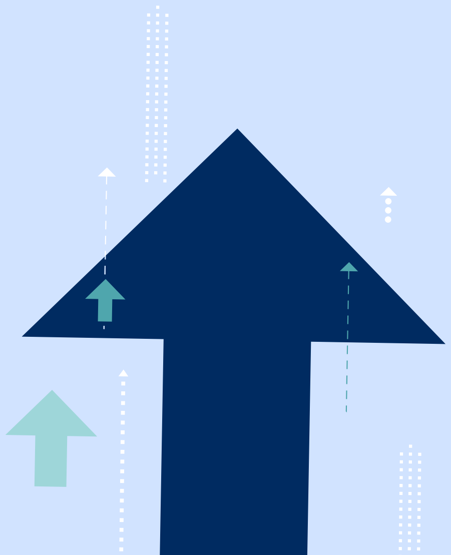
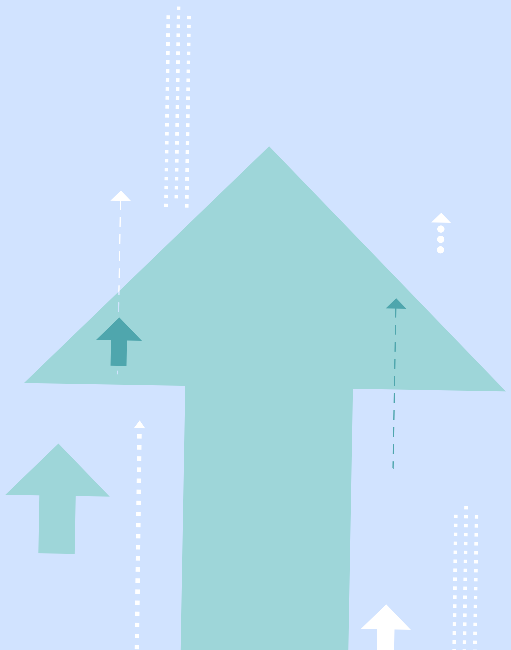
Agile integration architecture

Using lightweight integration runtimes to implement a container-based and microservices-aligned integration architecture

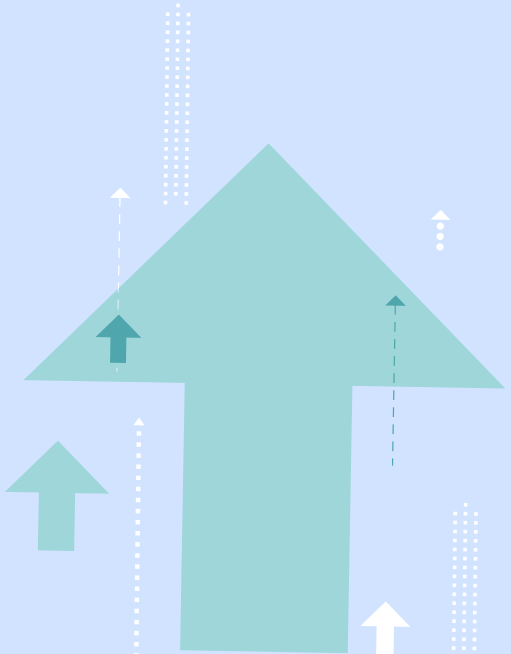
Contents:



Contents:



Contents:



Authors

Kim Clark
Integration Architect
kim.clark@uk.ibm.com

Kim is a technical strategist on IBMs integration portfolio working as an architect providing guidance to the offering management team on current trends and challenges. He has spent the last couple of decades working in the field implementing integration and process related solutions.

Tony Curcio
Director Application Integration
tcurcio@us.ibm.com

After years of implementing integration solutions in a variety of technologies, Tony joined the IBM offering management team in 2008. He now leads the Application Integration team in working with customers as they adopt more agile models for building integration solutions and embrace cloud as part of their IT landscape.

Nick Glowacki
Technical Specialist
nick.glowacki@ibm.com

Nick is a technical evangelist for IBMs integration portfolio working as a technical specialist exploring current trends and building leading edge solutions. He has spent the last 5 years working in the field and guiding a series of teams through their microservices journey. Before that he spent 5+ years in various other roles such as a developer, an architect and a IBM DataPower specialist. Over the course of his career he's been a user of node, xsl, JSON, Docker, Solr, IBM API Connect, Kubernetes, Java, SOAP, XML, WAS, Docker, Filenet, MQ, C++, CastIron, IBM App Connect, IBM Integration Bus.

THANK YOU

Sincere thanks go to the following people for their significant and detailed input and review of the material:

Carsten Bornert, Andy Garratt, Alan Glickenhouse, Rob Nicholson, Brian Petrini, Claudio Tagliabue, and Ben Thompson.

Executive Summary

The organization pursuing digital transformation must embrace new ways to use and deploy integration technologies, so they can move quickly in a manner appropriate to the goals of multicloud, decentralization and microservices. The application integration layer must transform to allow organizations to move boldly in building new customer experiences, rather than forcing models for architecture and development that pull away from maximizing the organization's productivity.

Many organizations have started embracing agile application techniques such as microservice architecture and are now starting to see the benefits of that shift. This approach complements and accelerates an enterprise's API strategy. Businesses should also seek to use this approach to modernize their existing ESB infrastructure to achieve more effective ways to manage and operate their integration services in their private or public cloud.

This book explores the merits of what we refer to as *agile integration architecture*[1] - a container-based, decentralized and microservice-aligned approach for integration solutions that meets the demands of agility, scalability and resilience required by digital transformation.

Agile integration architecture enables building, managing and operating effectively and efficiently to achieve the goals of digital transformation. It includes three distinct aspects that we will explore in detail:

a) Fine-grained integration deployment | b) Decentralized integration ownership and | c) Cloud-native integration infrastructure

[1]Note that we have used the term "lightweight integration" in the past, but have moved to the more appropriate "agile integration architecture".

How to navigate the book

The book is divided into three sections.

Section 1: The Impact of Digital Transformation on Integration

The impact of digital transformation

The rise of the digital economy, like most of the seismic technology shifts over the past several centuries, has fundamentally changed not only technology but business as well. The very concept of “digital economy” continues to evolve. Where once it was just a section of the economy that was built on digital technologies it has evolved becoming almost indistinguishable from the “traditional economy” and growing to include almost any new technology such as mobile, the Internet of Things, cloud computing, and augmented intelligence.

At the heart of the digital economy is the basic need to connect disparate data no matter where it lives. This has led to the rise of application integration, the need to connect multiple applications and data to deliver the greatest insight to the people and systems who can act on it. In this section we will explore how the digital economy created and then altered our concept of application integration.

- Chapter 1: **Integration has changed**
  Explores the effect that digital transformation has had on both the application and integration landscape, and the limitations of previous techniques.

- Chapter 2: **The journey so far: SOA, ESBs and APIs**
  Explores what led us up to this point, the pros and cons of SOA and the ESB pattern, the influence of APIs and the introduction of microservices architecture.

- Chapter 3: **The case for agile integration architecture**
  Explains how agile integration architecture exploits the principles of microservices architecture to address these new needs.

Chapter 1: Integration has changed

The impact of digital transformation

Over the last two years we've seen a tremendous acceleration in the pace that customers are establishing digital transformation initiatives. In fact, IDC estimates that digital transformation initiatives represent a $20 trillion market opportunity over the next 5 years. That is a staggering figure with respect to the impact across all industries and companies of all sizes. A primary focus of this digital transformation is to build new customer experiences through connected experiences across a network of applications that leverage data of all types.

However, bringing together these processes and information sources at the right time and within the right context has become increasingly complicated. Consider that many organizations have aggressively adopted SaaS business applications which have spread their key data sources across a much broader landscape. Additionally, new data sources that are available from external data providers must be injected into business processes to create competitive differentiation.

Finally, AI capabilities - which are being attached to many customer-facing applications - require a broad range of information to train, improve and correctly respond to business events. These processes and information sources need to be integrated by making them accessible synchronously via APIs, propagated in near real time by event streams, and a multitude of other mechanisms, more so than ever before.

It is no wonder that this growing complexity has increased the enterprise's need for and investment in integration capabilities. The pace of these investments, in both digital transformation generally and integration specifically, have led to a series of changes in how organizations are building solutions. Progressive IT shops have sought out, and indeed found, more agile ways to develop than were typical even just a few years ago.

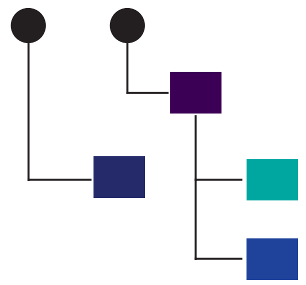
To drive new customer experiences organizations must tap into an ever-growing set of applications, processes and information sources – all of which significantly expand the enterprise's need for and investment in integration capabilities.

[2]IDC MaturityScape Benchmark: Digital Transformation Worldwide, 2017, Shawn Fitzgerald.

The value of application integration for digital transformation

When we consider the agenda for building new customer experiences and focus on how data is accessed and made available for the services and APIs that power these initiatives, we can clearly recognize several significant benefits that application integration brings to the table.

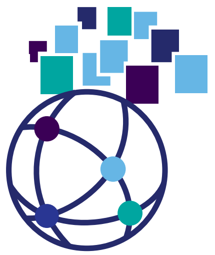
1. Effectively address disparity:



One of the key strengths of integration tooling is the ability to access data from any system with any sort of data in any sort of format and build homogeneity. The application landscape is only growing more diverse as organizations adopt SaaS applications and build new solutions in the cloud, spreading their data further across a hybrid set of systems. Even in the world of APIs, there are variations in data formats and structures that must be addressed.

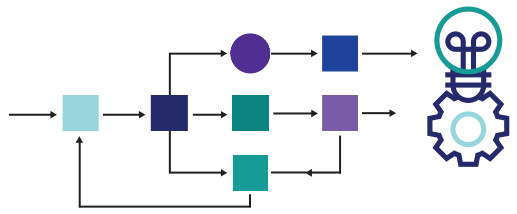
Furthermore, every system has subtleties in the way it enables updates, and surfaces events. The need for the organization to address information disparity is therefore growing at that same pace, and application integration must remain equipped to address the challenge of emerging formats.

2. Expertise of the endpoints:



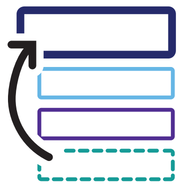
Each system has its own peculiarities that must be understood and responded to. Modern integration includes smarts around complex protocols and data formats, but it goes much further than that. It also incorporates intelligence about the actual objects, business and functions within the end systems. Application integration tooling is compassionate - understanding how to work with each system distinctly. This knowledge of the endpoint must include not only errors, but authentication protocols, load management, performance optimization, transactionality, idempotence, and much, much more. By including such features “in the box”, application integration yields tremendous gains in productivity over coding, and arguably a more consistent level of enterprise-class resiliency.

3. Innovation through data:



Applications in a digital world owe much of their innovation to their opportunity to combine data that is beyond their boundaries and create new meaning from it. This is particularly visible in microservices architecture, where the ability of application integration technologies to intelligently draw multiple sources of data together is often a core business requirement. Whether composing multiple API calls together or interpreting event streams, the main task of many microservices components is essentially integration.

4. Enterprise-grade artifacts:



Integration flows developed through application integration tooling inherit a tremendous amount of value from the runtime. Users can focus on building the business logic without having to worry about the surrounding infrastructure. The application integration runtime includes enterprise-grade features for error recovery, fault tolerance, log capture, performance analysis, message tracing, transactional update and recovery. Additionally, in some tools the artifacts are built using open standards and consistent best practices without requirements for the IT team to be experts in those domains.

Each of these factors (data disparity, expert endpoints, innovation through data, and enterprise grade artifacts) is causing a massive shift in how an integration architecture needs to be conceived, implemented and managed. The result is that organizations, and architects in particular, are reconsidering what integration means in the new digital age. Enter agile integration architecture, a container-based, decentralized and microservices-aligned approach for integration solutions that meets the demands of agility, scalability and resilience required by digital transformation.

The integration landscape is changing apace with enterprise and marketplace computing demands, but how did we get from SOA and ESBs to modern, containerized, agile integration architecture?

Application Integration benefits organizations building digital transformation solutions by effectively addressing information disparity, providing expert knowledge of application endpoints, easily orchestrating activities across applications, and lowering the cost of building expert-level artifacts.

Chapter 2: The journey so far: SOA, ESBs and APIs

Before we dive into agile integration architecture, we first need to understand what came before in a little more detail. In this chapter we will briefly look at the challenges of SOA by taking a closer look at what the ESB pattern was, how it evolved, where APIs came onto the scene, and the relationship between all that and microservices architecture.

Let’s start with SOA and the ESB and what went wrong.

The forming of the ESB pattern

As we started the millennium, we saw the beginnings of the first truly cross-platform protocol for interfaces. The internet, and with it HTTP, had become ubiquitous, XML was limping its way into existence off the back of HTML, and the SOAP protocols for providing synchronous web service interfaces were just taking shape. Relatively wide acceptance of these standards hinted at a brighter future where any system could discover and talk to any other system via a real-time synchronous remote procedure call, without reams of integration code as had been required in the past.

From this series of events, service-oriented architecture was born. The core purpose of SOA was to expose data and functions buried in systems of record over well-formed, simple-to-use, synchronous interfaces, such as web services. Clearly, SOA was about more than just providing those services, and often involved some significant re-engineering to align the back-end systems with the business needs, but the end goal was a suite of well-defined common re-usable services collating disparate systems. This would enable new applications to be implemented without the burden of deep integration every time, as once the integration was done for the first time and exposed as a service, it could be re-used by the next application.

However, this simple integration was a one-sided equation. We might have been able to standardize these protocols and data formats, but the back-end systems of record were typically old and had antiquated protocols and data formats for their current interfaces. Figure 1 below shows where the breakdown typically occurred. Something was needed to mediate between the old system and the new cross-platform protocols.

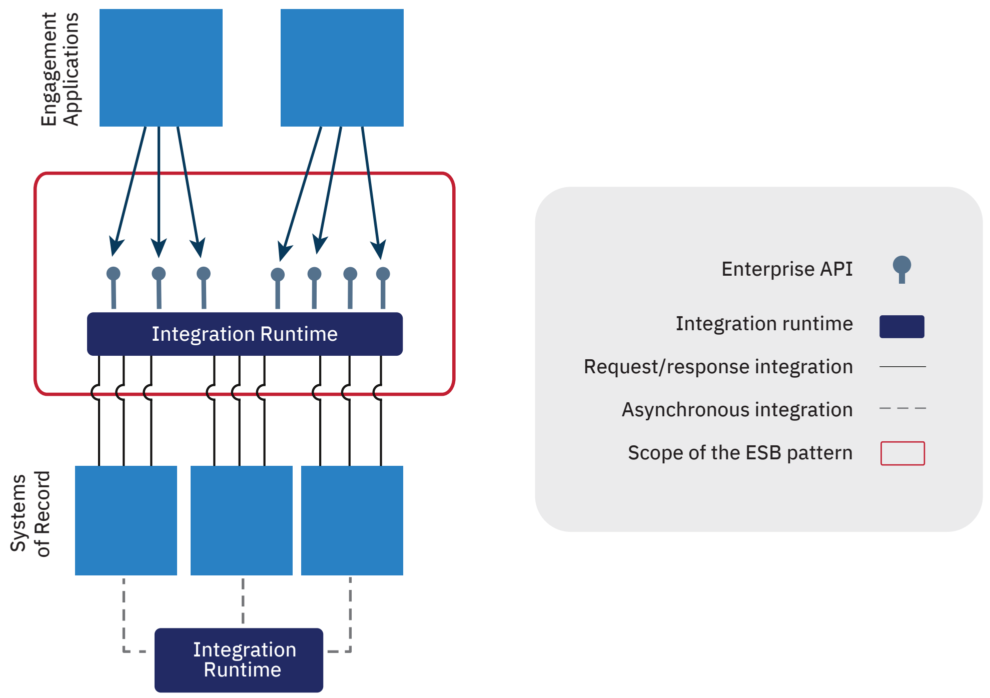


Figure 1. Synchronous centralized exposure pattern

This synchronous exposure pattern via web services was what the enterprise services bus (ESB) term was introduced for. It's all in the name—a centralized "bus" that could provide web "services" across the "enterprise". We already had the technology (the integration runtime) to provide connectivity to the back-end systems, coming from the preceding hub-and-spoke pattern. These integration runtimes could simply be taught to offer integrations synchronously via SOAP/HTTP, and we'd have our ESB.

What went wrong for the centralized ESB pattern?

While many large enterprises successfully implemented the ESB pattern, the term is often disparaged in the cloud-native space, and especially in relation to microservices architecture. It is seen as heavyweight and lacking in agility. What has happened to make the ESB pattern appear so outdated?

SOA turned out to be a little more complex than just the implementation of an ESB for a host of reasons—not the least of which was the question of who would fund such an enterprise-wide program. Implementing the ESB pattern itself also turned out to be no small task.

The ESB pattern often took the "E" in ESB very literally and implemented a single infrastructure for the whole enterprise, or at least one for each significant part of the enterprise. Tens or even hundreds of integrations might have been installed on a production server cluster, and if that was scaled up, they would be present on every clone within that cluster. Although this heavy centralization isn't required by the ESB pattern itself, it was almost always present in the resultant topology. There were good reasons for this, at least initially: hardware and software costs were shared, provisioning of the servers only had to be performed once, and due to the relative complexity of the software, only one dedicated team of integration specialists needed to be skilled up to perform the development work.

The centralized ESB pattern had the potential to deliver significant savings in integration costs if interfaces could be re-used from one project to the next (the core benefit proposition of SOA). However, coordinating such a cross-enterprise initiative and ensuring that it would get continued funding—and that the funding only applied to services that would be sufficiently re-usable to cover their creation costs—proved to be very difficult indeed. Standards and tooling were maturing at the same time as the ESB patterns were being implemented, so the implementation cost and time for providing a single service were unrealistically high.

ESB patterns have had issues ensuring continued funding for cross-enterprise initiatives since those do not apply specifically within the context of a business initiative.

Often, line-of-business teams that were expecting a greater pace of innovation in their new applications became increasingly frustrated with SOA, and by extension the ESB pattern.

Some of the challenges of a centralized ESB pattern were:

- Deploying changes could potentially destabilize other unrelated interfaces running on the centralized ESB.
- Servers containing many integrations had to be kept running and patched live wherever possible.

- Topologies for high availability and disaster recovery were complex and expensive.
- For stability, servers typically ran many versions behind the current release of software reducing productivity.
- The integration specialist teams often didn't know much about the applications they were trying to integrate with.
- Pooling of specialist integration skilled people resulted in more waterfall style engagement with application teams.
- Service discovery was immature so documentation became quickly outdated.

The result was that creation of services by this specialist SOA team became a bottleneck for projects rather than the enabler that it was intended to be. This typically gave by association the centralized ESB pattern a bad name.

Formally, as we've described, ESB is an architectural pattern that refers to the exposure of services. However, as mentioned above, the term is often over-simplified and applied to the integration engine that's used to implement the pattern. This erroneously ties the static and aging centralized ESB pattern with integration engines that have changed radically over the intervening time.

Integration engines of today are significantly more lightweight, easier to install and use, and can be deployed in more decentralized ways that would have been unimaginable at the time the ESB concept was born. As we will see, agile integration architecture enables us to overcome the limitations of the ESB pattern.

If you would like a deeper introduction into where the ESB pattern came from and a detailed look at the benefits, and the challenges that came with it, take a look at the source material for this section in the following article:

http://ibm.biz/FateOfTheESBPaper

The API economy and bi-modal IT

External APIs have become an essential part of the online persona of many companies, and are at least as important as its websites and mobile applications. Let's take a brief look at how that evolved from the maturing of internal SOA based services.

SOAP-style RPC interfaces proved complex to understand and use, and simpler and more consistent RESTful services provided using JSON/HTTP became a popular mechanism.

But the end goal was the same: to make functions and data available via standardized interfaces so that new applications could be built on top of them more quickly.

With the broadening usage of these service interfaces, both within and beyond the enterprise, more formal mechanisms for providing services were required. It quickly became clear that simply making something available over a web service interface, or latterly as a RESTful JSON/HTTP API, was only part of the story.

That service needed to be easily discovered by potential consumers, who needed a path of least resistance for gaining access to it and learning how to use it. Additionally, the providers of the service or API needed to be able to place controls on its usage, such as traffic control and an appropriate security model. Figure 2 below demonstrates how the introduction of service/API gateways effects the scope of the ESB pattern.

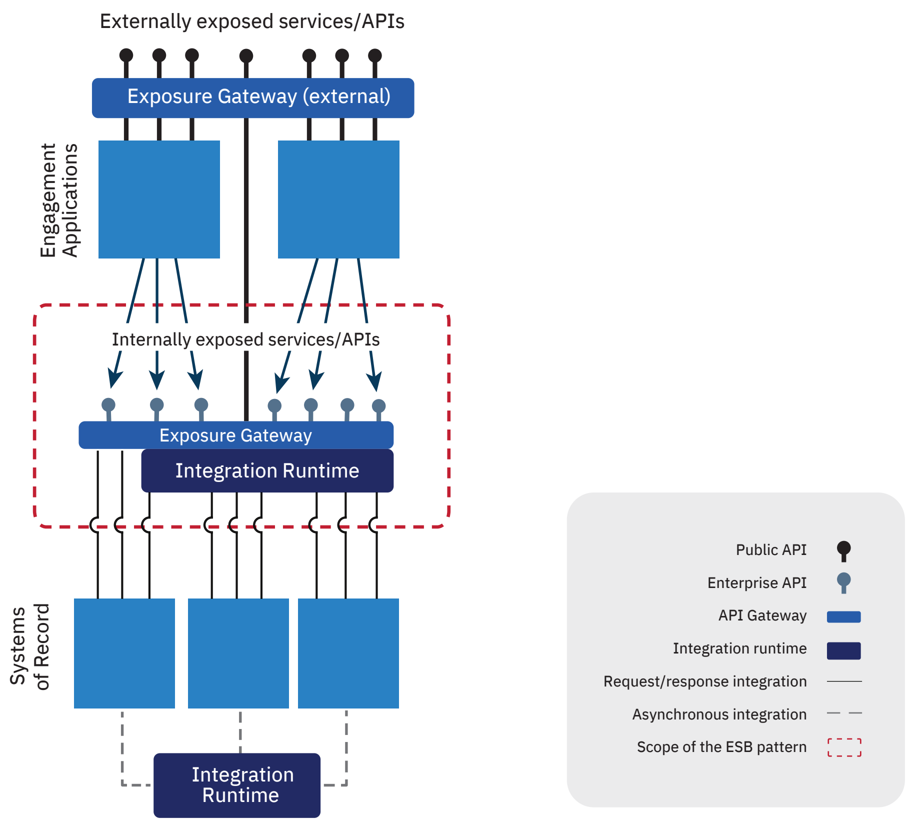


Figure 2. Introduction of service/API gateways internally and externally

The typical approach was to separate the role of service/API exposure out into a separate gateway. These capabilities evolved into what is now known as API management and enabled simple administration of the service/API. The gateways could also be specialized to focus on API management-specific capabilities, such as traffic management (rate/throughput limiting), encryption/decryption, redaction, and security patterns. The gateways could also be supplemented with portals that describe the available APIs which enable self-subscription to use the APIs along with provisioning analytics for both users and providers of the APIs.

While logically, the provisioning of APIs outside the enterprise looks like just an extension of the ESB pattern, there are both significant infrastructural and design differences between externally facing APIs and internal services/APIs.

- From an infrastructural point of view, it is immediately obvious that the APIs are being used by consumers and devices that may exist anywhere from a geographical and network point of view. As a result, it is necessary to design the APIs differently to take into account the bandwidth available and the capabilities of the devices used as consumers.

- From a design perspective, we should not underestimate the difference in the business objectives of these APIs. External APIs are much less focused on re-use, in the way that internal APIs/ services were in SOA, and more focused on creating services targeting specific niches of potential for new business. Suitably crafted channel specific APIs provide an enterprise with the opportunity to radically broaden the number of innovation partners that it can work with (enabling crowd sourcing of new ideas), and they play a significant role in the disruption of industries that is so common today. This realization caused the birth of what we now call the API Economy, and it is a well-covered topic on IBMs "API Economy" blog.

The main takeaway here is that this progression exacerbated an already growing divide between the older traditional systems of record that still perform all the most critical transactions fundamental to the business, and what became known as the systems of engagement, where innovation occurred at a rapid pace, exploring new ways of interacting with external consumers. This resulted in **bi-modal IT**, where new decentralized, fast-moving areas of IT needed much greater agility in their development and led to the invention of new ways of building applications using, for example, microservices architecture.

The rise of lightweight runtimes

Earlier, we covered the challenges of the heavily centralized integration runtime—hard to safely and quickly make changes without affecting other integrations, expensive and complex to scale, etc.

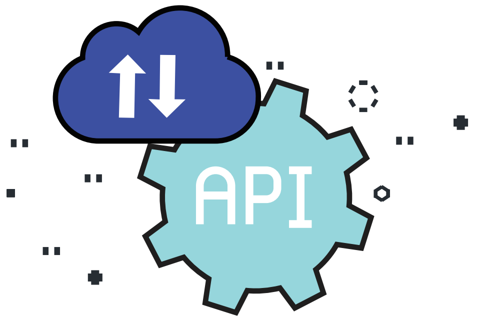
Sound familiar? It should. These were exactly the same challenges that application development teams were facing at the same time: bloated, complex application servers that contained too much interconnected and cross-dependent code, on a fragile cumbersome topology that was hard to replicate or scale. Ultimately, it was this common paradigm that led to the emergence of the principles of microservices architecture. As lightweight runtimes and application servers such as Node.js and IBM WAS Liberty were introduced—runtimes that started in seconds and had tiny footprints—it became easier to run them on smaller virtual machines, and then eventually within container technologies such as Docker.

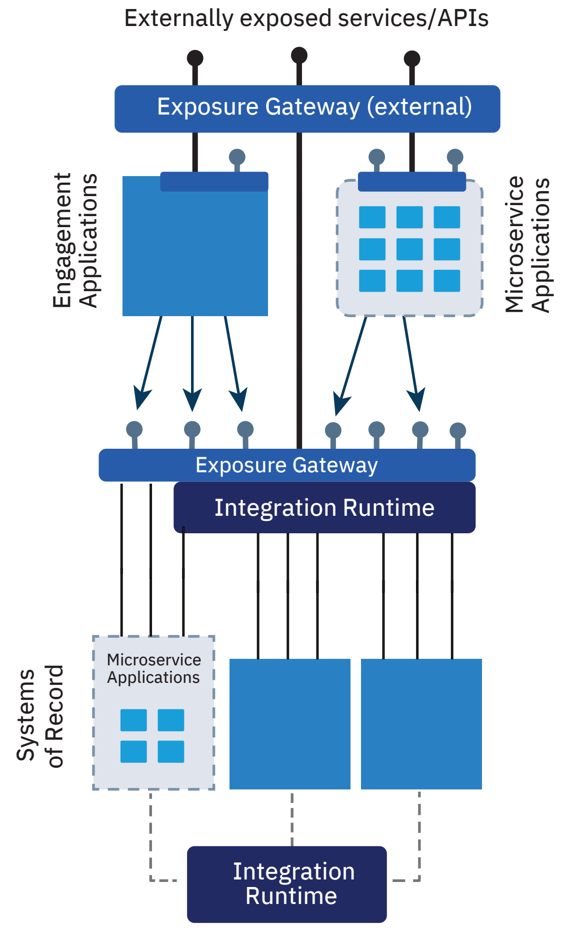
Microservices architecture: A more agile and scalable way to build applications

In order to meet the constant need for IT to improve agility and scalability, a next logical step in application development was to break up applications into smaller pieces and run them completely independently of one another. Eventually, these pieces became small enough that they deserved a name, and they were termed **microservices**.

If you take a closer look at microservices concepts, you will see that it has a much broader intent than simply breaking things up into smaller pieces. There are implications for architecture, process, organization, and more—all focused on enabling organizations to better use cloud-native technology advances to increase their pace of innovation.

However, focusing back on the core technological difference, these small independent microservices components can be changed in isolation to create greater agility, scaled individually to make better use of cloud-native infrastructure, and managed more ruthlessly to provide the resilience required by 24/7 online applications. Figure 3 below visualizes the microservices architecture we've just described.




Externally exposed services/APIs
Exposure Gateway (external)
Engagement Applications
Microservice Applications
Exposure Gateway
Integration Runtime
Systems of Record
Microservice Applications
Integration Runtime

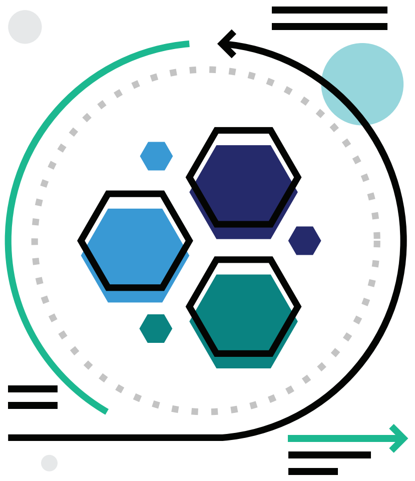
Public API	●
Enterprise API	●
API Gateway	▬
Lightweight language runtime	■
Integration runtime	▬
Request/response integration	——
Asynchronous integration	– – –
Microservice application boundary	⬚

Figure 3. Microservices architecture: A new way to build applications

Not least is your challenge of deciding the shape and size of your microservices components. Add to that equally critical design choices around the extent to which you decouple them. You need to constantly balance practical reality with aspirations for microservices-related benefits. *In short, your microservices-based application is only as agile and scalable as your design is good, and your methodology is mature.*

In theory, these principles could be used anywhere. Where we see them most commonly is in the systems of engagement layer, where greater agility is essential. However, they could also be used to improve the agility, scalability, and resilience of a system of record—or indeed anywhere else in the architecture, as you will see as we discuss agile integration architecture in more depth.

Without question, microservices principles can offer significant benefits under the right circumstances. However, choosing the right time to use these techniques is critical, and getting the design of highly distributed components correct is not a trivial endeavor.



A comparison of SOA and microservice architecture

Microservices inevitably gets compared to SOA in architectural discussions, not least because they share many words in common. However, as you will see, this comparison is misleading at best, since the terms apply to two very different scopes. Figure 4 demonstrates how microservices are application-scoped within the SOA enterprise service bus.

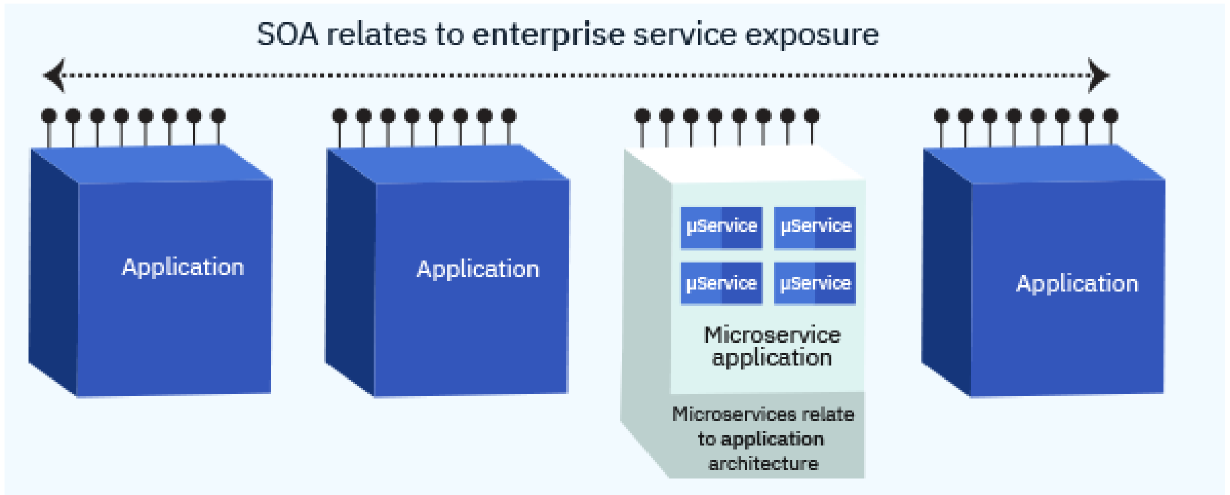

SOA relates to enterprise service exposure
Application
Application
μService
μService
μService
μService
Microservice application
Microservices relate to application architecture
Application

Figure 4. SOA is enterprise scoped, microservices architecture is application scoped

Service-oriented architecture is an enterprise-wide initiative to create re-usable, synchronously available services and APIs, such that new applications can be created more quickly incorporating data from other systems.

Microservices architecture, on the other hand, is an option for how you might choose to write an individual application in a way that makes that application more agile, scalable, and resilient.

Service-oriented architecture is an enterprise-wide initiative.

Microservices architecture is an option for how you might choose to write an individual application.

It's critical to recognize this difference in scope, since some of the core principles of each approach could be completely incompatible if applied at the same scope. For example:

- **Re-use:** In SOA, re-use of integrations is the primary goal, and at an enterprise level, striving for some level of re-use is essential. In microservices architecture, creating a microservices component that is re-used at runtime throughout an application results in dependencies that reduce agility and resilience. Microservices components generally prefer to re-use code by copy and accept data duplication to help improve decoupling between one another.

- **Synchronous calls:** The re-usable services in SOA are available across the enterprise using predominantly synchronous protocols such as RESTful APIs. However, within a microservice application, synchronous calls introduce real-time dependencies, resulting in a loss of resilience, and also latency, which impacts performance. Within a microservices application, interaction patterns based on asynchronous communication are preferred, such as event sourcing where a publish subscribe model is used to enable a microservices component to remain up to date on changes happening to the data in another component.

- **Data duplication:** A clear aim of providing services in an SOA is for all applications to synchronously get hold of, and make changes to, data directly at its primary source, which reduces the need to maintain complex data synchronization patterns. In microservices applications, each microservice ideally has local access to all the data it needs to ensure its independence from other microservices, and indeed from other applications—even if this means some duplication of data in other systems. Of course, this duplication adds complexity, so it needs to be balanced against the gains in agility and performance, but this is accepted as a reality of microservices design.

So, in summary, SOA has an enterprise scope and looks at how integration occurs between applications. Microservices architecture has an application scope, dealing with how the internals of an application are built. This is a relatively swift explanation of a much more complex debate, which is thoroughly explored in a separate article:
http://ibm.biz/MicroservicesVsSoa

However, we have enough of the key concepts to now delve into the various aspects of agile integration architecture.

Chapter 3: The case for agile integration architecture

Let’s briefly explore why microservices concepts have become so popular in the application space. We can then quickly see how those principles can be applied to the modernization of integration architecture.

greater Agility

They are small enough to be understood completely in isolation and changed independently

elastic Scalability

Their resource usage can be truly tied to the business model

discrete Resilience

With suitable decoupling, changes to one microservice do not affect others at runtime

Microservices architecture

Microservices architecture is an alternative approach to structuring applications. Rather than an application being a large silo of code all running on the same server, an application is designed as a collection of smaller, completely independently running components. This enables the following benefits, which are also illustrated in Figure 5 below:

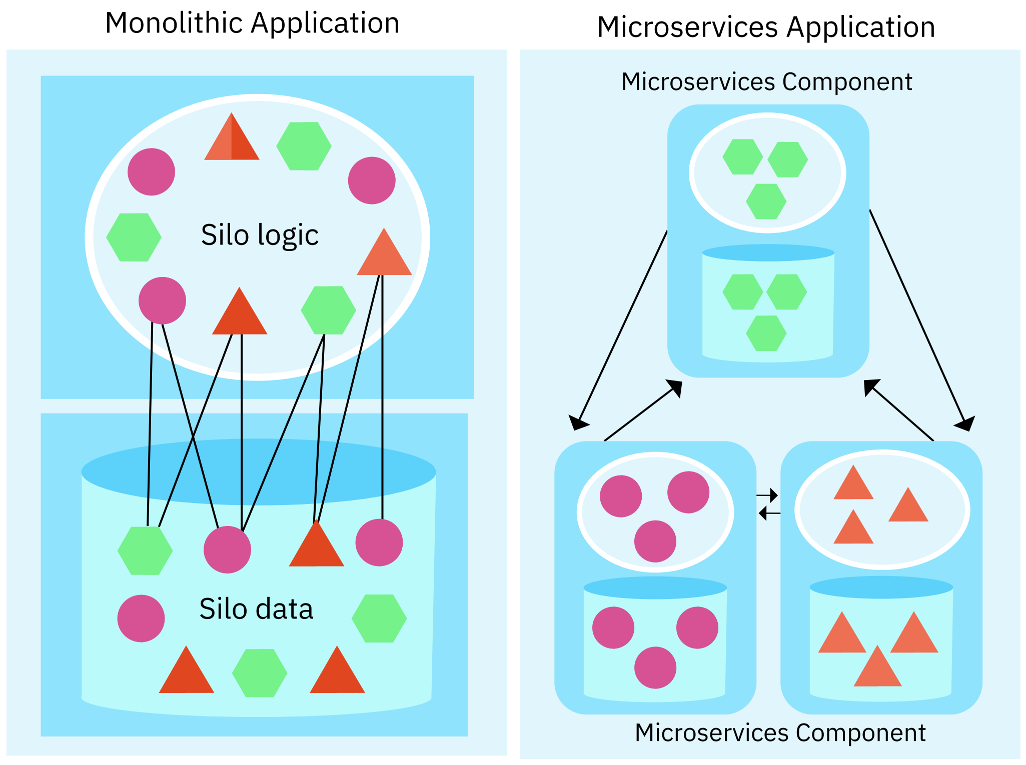


Figure 5 Comparison of siloed and microservices-based applications

Microservice components are often made from pure language runtimes such as Node.js or Java, but equally they can be made from any suitably lightweight runtime. The key requirements include that they have a simple dependency-free installation, file system based deploy, start/stop in seconds and have strong support for container-based infrastructure.

Microservices architectures lead to the primary benefits of greater agility, elastic scalability, and discrete resilience.

Microservices architecture enables developers to make better use of cloud native infrastructure and manage components more ruthlessly, providing the resilience and scalability required by 24/7 online applications. It also improves ownership in line with DevOps practices whereby a team can truly take responsibility for a whole microservice component throughout its lifecycle and hence make changes at a higher velocity.

As with any new approach there are challenges too, some obvious, and some more subtle. Microservices are a radically different approach to building applications. Let's have a brief look at some of the considerations:

- **Greater overall complexity:** Although the individual components are potentially simpler, and as such they are easier to change and scale, the overall application is inevitably a collection of highly distributed individual parts.
- **Learning curve on cloud-native infrastructure:** To manage the increased number of components, new technologies and frameworks are required including service discovery, workload orchestration, container management, logging frameworks and more. Platforms are available to make this easier, but it is still a learning curve.
- **Different design paradigms:** The microservices application architecture requires fundamentally different approaches to design. For example, using eventual consistency rather than transactional interactions, or the subtleties of asynchronous communication to truly decouple components.
- **DevOps maturity:** Microservices require a mature delivery capability. Continuous integration, deployment, and fully automated tests are a must. The developers who write code must be responsible for it in production. Build and deployment chains need significant changes to provide the right separation of concerns for a microservices environment.

Microservices architecture is not the solution to every problem. Since there is an overhead of complexity with the microservices approach, it is critical to ensure the benefits outlined above outweigh the extra complexity. However, if applied judiciously it can provide order of magnitude benefits that would be hard to achieve any other way.

Microservices architecture discussions are often heavily focused on alternate ways to build applications, but the core ideas behind it are relevant to all software components, including integration.

Agile integration architecture

If what we've learned from microservices architecture means it sometimes makes sense to build applications in a more granular lightweight fashion, why shouldn't we apply that to integration to?

Integration is typically deployed in a very siloed and centralized fashion such as the ESB pattern. What would it look like if we were to re-visit that in the light of microservices architecture? It is this alternative approach that we call **"agile integration architecture"**.

Agile integration architecture is defined as

"a container-based, decentralized and microservices-aligned architecture for integration solutions".

There are three related, but separate aspects to agile integration architecture:

- **Aspect 1: Fine-grained integration deployment.**

  What might we gain by breaking out the integrations in the siloed ESB into separate runtimes?

- **Aspect 2: Decentralized integration ownership.**

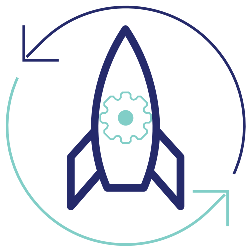
  How should we adjust the organizational structure to better leverage a more fine-grained approach?

- **Aspect 3: Cloud native integration infrastructure.**

  What further benefits could we gain by a fully cloud-native approach to integration.

Although these each have dedicated chapters, it's worth taking the time to summarize them at a conceptual level here.

Aspect 1: Fine-grained integration deployment



The centralized deployment of integration hub or enterprise services bus (ESB) patterns where all integrations are deployed to a single heavily nurtured (HA) pair of integration servers has been shown to introduce a bottleneck for projects. Any deployment to the shared servers runs the risk of destabilizing existing critical interfaces. No individual project can choose to upgrade the version of the integration middleware to gain access to new features.

We could break up the enterprise-wide ESB component into smaller more manageable and dedicated pieces. Perhaps in some cases we can even get down to one runtime for each interface we expose.

These “fine-grained integration deployment” patterns provide specialized, right-sized containers, offering improved agility, scalability and resilience, and look very different to the centralized ESB patterns of the past. Figure 6 demonstrates in simple terms how a centralized ESB differs from fine-grained integration deployment.B patterns of the past.

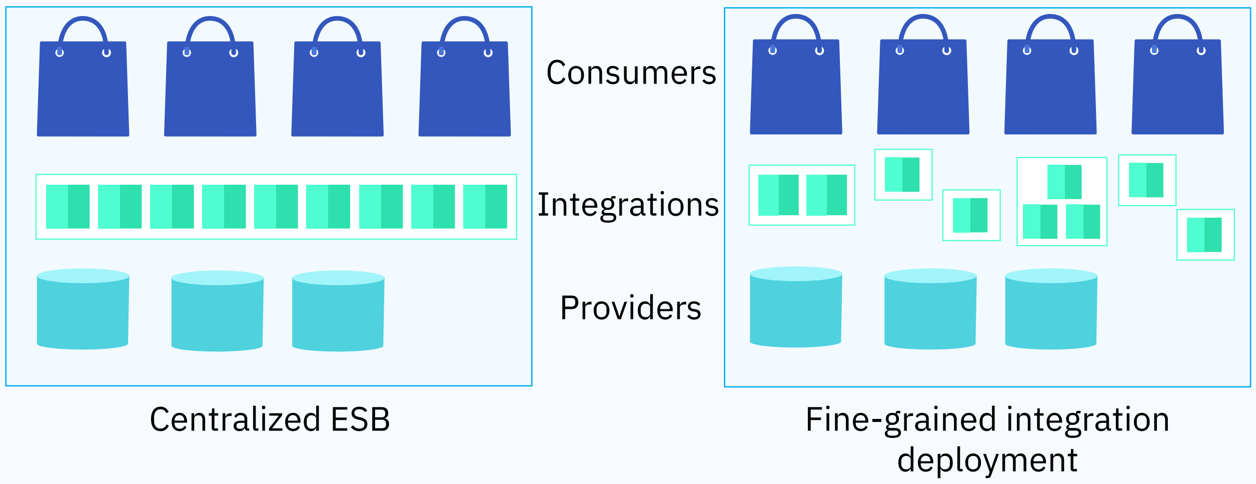


Figure 6: Simplistic comparison of a centralized ESB to fine-grained integration deployment

Fine-grained integration deployment draws on the benefits of a microservices architecture we listed in the last section: agility, scalability and resilience:

Agility:

Different teams can work on integrations independently without deferring to a centralized group or infrastructure that can quickly become a bottleneck. Individual integration flows can be changed, rebuilt, and deployed independently of other flows, enabling safer application of changes and maximizing speed to production.

Scalability:

Individual flows can be scaled on their own, allowing you to take advantage of efficient elastic scaling of cloud infrastructures.

Resilience:

Isolated integration flows that are deployed in separate containers cannot affect one another by stealing shared resources, such as memory, connections, or CPU.

Breaking the single ESB runtime up into many separate runtimes, each containing just a few integrations is explored in detail in “Chapter 4: **Aspect 1:** Fine grained integration deployment”

Aspect 2:
Decentralized integration ownership

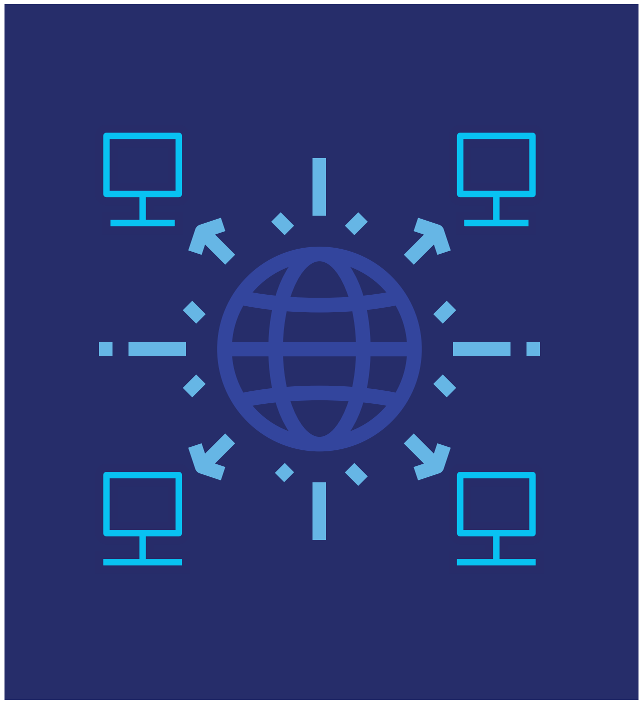
A significant challenge faced by service-oriented architecture was the way that it tended to force the creation of central integration teams, and infrastructure to create the service layer.

This created ongoing friction in the pace at which projects could run since they always had the central integration team as a dependency. The central team knew their integration technology well, but often didn’t understand the applications they were integrating, so translating requirements could be slow and error prone.

Many organizations would have preferred the application teams own the creation of their own services, but the technology and infrastructure of the time didn’t enable that.

The move to fine-grained integration deployment opens a door such that ownership of the creation and maintenance of integrations can be distributed.

It’s not unreasonable for business application teams to take on integration work, streamlining the implementation of new capabilities. This shift is discussed in more depth in “Chapter 5: **Aspect 2:** Decentralized integration ownership”.



Aspect 3: Cloud-native integration infrastructure

Integration runtimes have changed dramatically in recent years. So much so that these lightweight runtimes can be used in truly cloud-native ways. By this we are referring to their ability to hand off the burden of many of their previously proprietary mechanisms for cluster management, scaling, availability and to the cloud platform in which they are running.

This entails a lot more than just running them in a containerized environment. It means they have to be able to function as “cattle not pets,” making best use of the orchestration capabilities such as Kubernetes and many other common cloud standard frameworks.

We expand considerably on the concepts in “Chapter 6: **Aspect 3:** Cloud native integration infrastructure”.

How has the modern integration runtime changed to accommodate agile integration architecture?

Clearly, agile integration architecture requires that the integration topology be deployed very differently. A key aspect of that is a modern integration runtime that can be run in a container-based environment and is well suited to cloud-native deployment techniques. Modern integration runtimes are almost unrecognizable from their historical peers. Let’s have a look at some of those differences:

- **Fast lightweight runtime:** They run in containers such as Docker and are sufficiently lightweight that they can be started and stopped in seconds and can be easily administered by orchestration frameworks such as Kubernetes.
- **Dependency free:** They no longer require databases or message queues, although obviously, they are very adept at connecting to them if they need to.
- **File system based installation:** They can be installed simply by laying their binaries out on a file system and starting them up-ideal for the layered file systems of Docker images.
- **DevOps tooling support:** The runtime should be continuous integration and deployment-ready. Script and property file-based install, build, deploy, and configuration to enable “infrastructure as code” practices. Template scripts for standard build and deploy tools should be provided to accelerate inclusion into DevOps pipelines.
- **API-first:** The primary communication protocol should be RESTful APIs. Exposing integrations as RESTful APIs should be trivial and based upon common conventions such as the Open API specification. Calling downstream RESTful APIs should be equally trivial, including discovery via definition files.
- **Digital connectivity:** In addition to the rich enterprise connectivity that has always been provided by integration runtimes, they must also connect to modern resources.

For example, NoSQL databases (MongoDb and Cloudant etc.), and Messaging services such as Kafka. Furthermore, they need access to a rich catalogue of application intelligent connectors for SaaS (software as a service) applications such as Salesforce.

- **Continuous delivery:** Continuous delivery is enabled by command-line interfaces and template scripts that mesh into standard DevOps pipeline tools. This further reduces the knowledge required to implement interfaces and increases the pace of delivery.
- **Enhanced tooling:** Enhanced tooling for integration means most interfaces can be built by configuration alone, often by individuals with no integration background. With the addition of templates for common integration patterns, integration best practices are burned into the tooling, further simplifying the tasks. Deep integration specialists are less often required, and some integration can potentially be taken on by application teams as we will see in the next section on decentralized integration.

Modern integration runtimes are well suited to the three aspects of agile integration architecture: fine-grained deployment, decentralized ownership, and true cloud-native infrastructure. Before we turn our attention to these aspects in more detail, we will take a more detailed look at the SOA pattern for those who may be less familiar with it, and explore where organizations have struggled to reach the potential they sought.

Section 2: Exploring agile integration architecture in detail

Now that you have been introduced to the concept of agile integration architecture we are going to dive into greater detail on its three main aspects, looking at their characteristics and presenting a real-life scenario.

- Chapter 4:

  Aspect 1: Fine-grained integration deployment
  Addresses the benefits an organization gains by breaking up the centralized ESB

- Chapter 5:

  Aspect 2: Decentralized integration ownership
  Discusses how shifting from a centralized governance and development practice creates new levels of agility and innovation.

- Chapter 6:

  Aspect 3: Cloud native integration infrastructure
  Provides a description of how adopting key technologies and practices from the cloud native application discipline can provide similar benefits to application integration.

Chapter 4: Aspect 1: Fine-grained integration deployment

If the large centralized ESB pattern containing all the integrations for the enterprise is reducing agility for all the reasons noted previously, then why not break it up into smaller pieces? This section explores why and how we might go about doing that.

Breaking up the centralized ESB

If it makes sense to build applications in a more granular fashion, why shouldn't we apply this idea to integration, too? We could break up the enterprise-wide centralized ESB component into smaller, more manageable, dedicated components. Perhaps even down to one integration runtime for each interface we expose, although in many cases it would be sufficient to bunch the integrations as a handful per component.

Figure 7 shows the result of breaking up the ESB into separate, independently maintainable and scalable components.

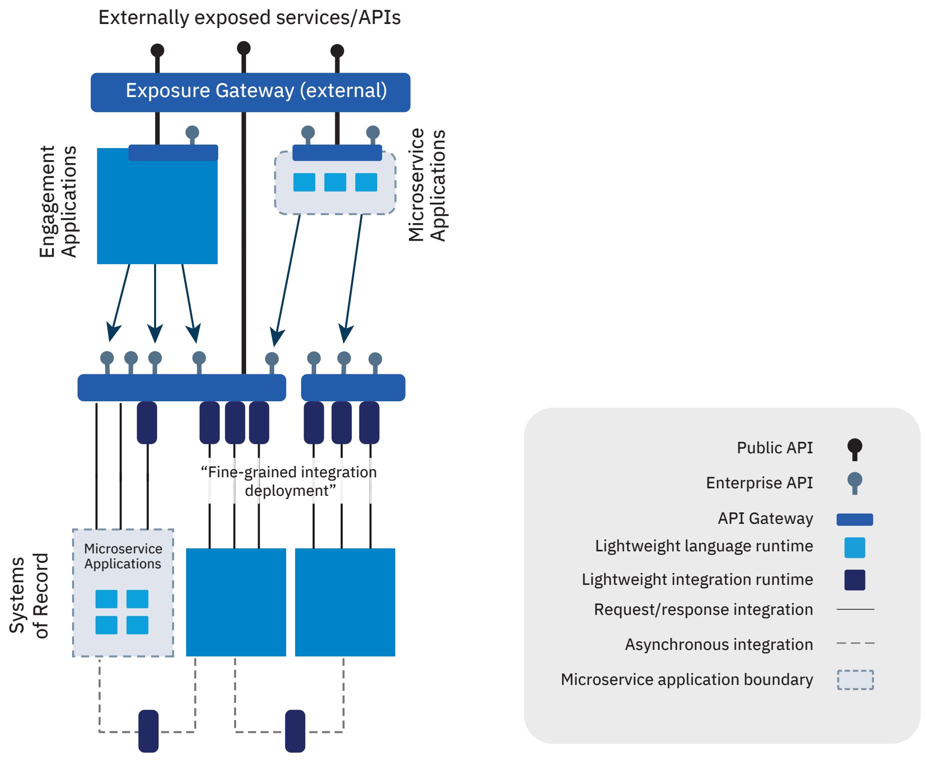

Externally exposed services/APIs
Exposure Gateway (external)
Engagement Applications
Microservice Applications
"Fine-grained integration deployment"
Systems of Record
Microservice Applications
Public API
Enterprise API
API Gateway
Lightweight language runtime
Lightweight integration runtime
Request/response integration
Asynchronous integration
Microservice application boundary

Figure 7: Breaking up the centralized ESB into independently maintainable and scalable pieces

The heavily centralized ESB pattern can be broken up in this way, and so can the older hub and spoke pattern. This makes each individual integration easier to change independently, and improves agility, scaling, and resilience.

Fine grained integration deployment allows you to make a change to an individual integration with complete confidence that you will not introduce any instability into the environment

This approach allows you to make a change to an individual integration with complete confidence that you will not introduce any instability into the environment on which the other integrations are running. You could choose to use a different version of the integration runtime, perhaps to take advantage of new features, without forcing a risky upgrade to all other integrations. You could scale up one integration completely independently of the others, making extremely efficient use of infrastructure, especially when using cloud-based models.

There are of course considerations to be worked through with this approach, such as the increased complexity with more moving parts. Also, although the above could be achieved using virtual machine technology, it is likely that the long-term benefits would be greater if you were to use containers such as Docker, and orchestration mechanisms such as Kubernetes. Introducing new technologies to the integration team can add a learning curve. However, these are the same challenges that an enterprise would already be facing if they were exploring microservices architecture in other areas, so that expertise may already exist within the organization.

We typically call this pattern fine-grained integration deployment (and a key aspect of agile integration architecture), to differentiate it from more purist microservices application architectures. We also want to mark a distinction from the ESB term, which is strongly associated with the more cumbersome centralized integration architecture.

What characteristics does the integration runtime need?

To be able to be used for fine-grained deployment, what characteristics does a modern integration runtime need?

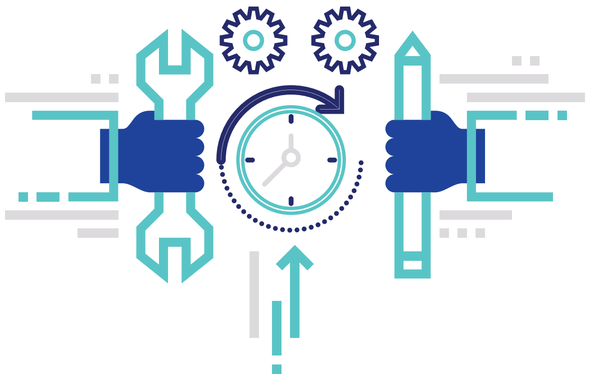
- **Fast, light integration runtime.**

  The actual runtime is slim, dispensing with hard dependencies on other components such as databases for configuration, or being fundamentally reliant on a specific message queuing capability. The runtime itself can now be stopped and started in seconds, yet none of its rich functionality has been sacrificed. It is totally reasonable to consider deploying a small number of integrations on a runtime like this and then running them independently rather than placing all integration on a centralized single topology.

  Installation is equally minimalist and straightforward requiring little more than laying binaries out on a file system.

- **Virtualization and containerization.**

  The runtime should actively support containerization technologies such as Docker and container orchestration capabilities such as Kubernetes, enabling non-functional characteristics such as high availability and elastic scalability to be managed in the standardized ways used by other digital generation runtimes, rather than relying on proprietary topologies and technology. This enables new runtimes to be introduced administered and scaled in well-known ways without requiring proprietary expertise.



- **Stateless**

  The runtime needs to able to run statelessly. In other words, runtimes should not be dependent on, or even aware of one another. As such they can be added and taken away from a cluster freely and new versions of interfaces can be deployed easily. This enables the container orchestration to manage scaling, rolling deployments, A/B testing, canary tests and more with no proprietary knowledge of the underlying integration runtime. This stateless aspect is essential if there are going to be more runtimes to manage in total.

- **Cloud-first**

  It should be possible to immediately explore a deployment without the need to install any local infrastructure. Examples include providing a cloud based managed service whereby integrations can be immediately deployed, with a low entry cost, and an elastic cost model. Quick starts should be available for simple creation of deployment environments on major cloud vendors' infrastructures.

This provides a taste of how different the integration runtimes of today are from those of the past.

IBM App Connect Enterprise (formerly known as IBM Integration Bus) is a good example of such a runtime. Integration runtimes are not in themselves an ESB; ESB is just one of the patterns they can be used for. They are used in a variety of other architectural patterns too, and increasingly in fine-grained integration deployment.

Granularity

A glaring question then remains: how granular should the decomposition of the integration flows be? Although you could potentially separate each integration into a separate container, it is unlikely that such a purist approach would make sense. The real goal is simply to ensure that unrelated integrations are not housed together. That is, a middle ground with containers that group related integrations together (as shown in Figure 8) can be sufficient to gain many of the benefits that were described previously.

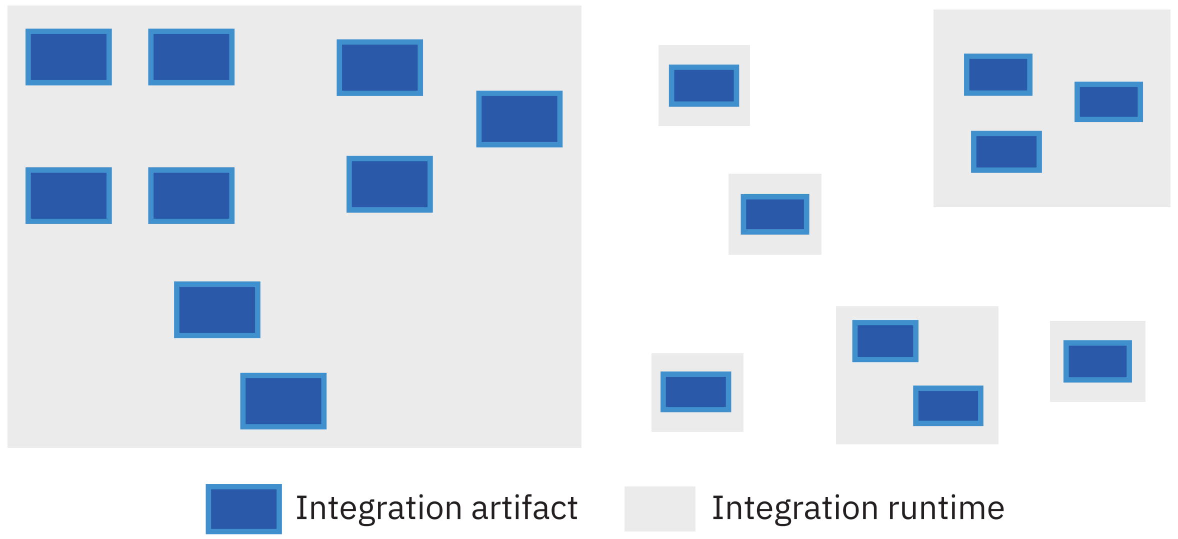


Figure 8: Related integrations grouped together can lead to many benefits.

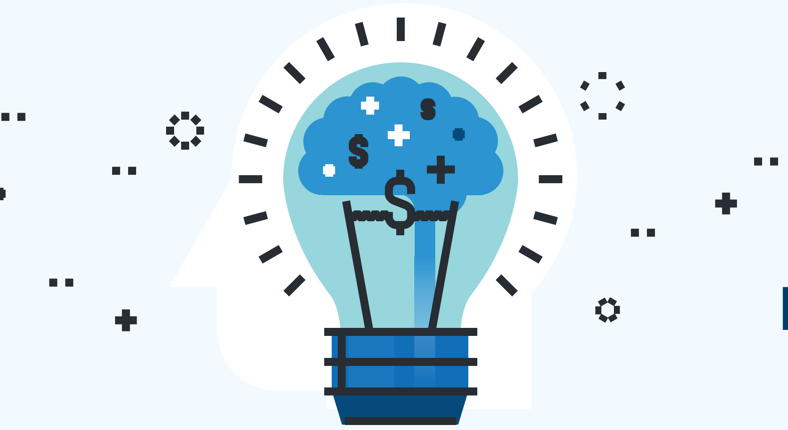
You target the integrations that need the most independence and break them out on their own. On the flip side, keep together flows that, for example, share a common data model for cross-compatibility. In a situation where changes to one integration must result in changes to all related integrations, the benefits of separation may not be so relevant.

For example, where any change to a shared data model must be performed on all related integrations, and they would all need to be regression tested anyway, having them as separate entities may only be of minimal value. However, if one of those related integrations has a very different scaling profile, there might be a case for breaking it out on its own. It's clear that there will always be a mixture of concerns to consider when assessing granularity.

The right level of granularity is to allow decomposition of the integration flows to the point where unrelated integrations are not housed together.

Conclusion on fine-grained integration deployment

Fine-grained deployment allows you to reap some of the benefits of microservices architecture in your integration layer enabling greater agility because of infrastructural decoupled components, elastic scaling of individual integrations and an inherent improvement in resilience from the greater isolation.



Lessons Learned

A real-life scenario

Let’s examine an organization where an agile methodology was adopted, a cloud had been chosen but who still had a centralized team that maintained an enterprise-wide data model and ESB. This team realized that they struggled with even a simple change of adding a new element to the enterprise message model and the associated exposed endpoint.

The team that owned the model took requests from application development teams. Since it wasn’t reasonable for the modelling CoE (Center of Excellence) team to take requests constantly, they met once a week to talk about changes and determine if the changes would be agreed to. To reduce change frequency, the model was released once a week with whatever updates had been accepted by the CoE. After the model was changed the ESB team would take action on any related changes. Because of the enterprise nature of the ESB this would then again have to be coordinated with other builds, other application needs and releases.

The problem

While this seemed like a reasonable approach, it created issues with the application development team. Adding one element to the model took, at best, two weeks. The application team had to submit the request, then attend the CoE meeting, then if agreed to that model would be released the following week. From there, the application dev team would get the model which would contain their change (and any other change any other team had submitted for between their last version and the current version). Then would be able to start work implementing business code.

After some time, these two week procedural delays began to add up. From this point we need to strongly consider if the value of the highly-governed, enterprise message model is worth that investment, and if the consistency gained through the CoE team is worth the delays. On the benefit side the CoE team can now create and maintain standards and keep a level of consistency, on the con side that consistency is incurring a penalty if we look at it from the lens of time to market.

The solution

The solution was to break the data model into bounded contexts based on business focus areas. Furthermore the integrations were divided up into groups based on those bounded contexts too, each running on separate infrastructure. This allowed each data model and its associated integrations to evolve independently as required yet still providing consistency across a now more narrow bounded context. It is worth noting that although this provided improved autonomy with regard to data model changes, the integration team were still separate from the application teams, creating scheduling and requirements handover latencies.
In the next section, we will discuss the importance of exploring changes to the organizational boundaries too.

Chapter 5: Aspect 2: Decentralized integration ownership

We can take what we've done in "Aspect 1: Fine grained integration deployment" a step further. If you have broken up the integrations into separate decoupled pieces, you may opt to distribute those pieces differently from an ownership and administration point of view as well.

The microservices approach encourages teams to gain increasing autonomy such that they can make changes confidently at a more rapid pace. When applied to integration, that means allowing the creation and maintenance of integration artifacts to be owned directly by application teams rather than by a single separate centralized team. This distribution of ownership is often referred to under the broader topic of "decentralization" which is a common theme in microservices architecture.

It is extremely important to recognize that decentralization is a significant change for most organizations. For some, it may be too different to take on board and they may have valid reasons to remain completely centrally organized. For large organizations, it is unlikely it will happen consistently across all domains. It is much more likely that only specific pockets of the organization will move to this approach - where it suits them culturally and helps them meet their business objectives.

We'll discuss what effect that shift would have on an organization, and some of the pros and cons of decentralization.

Decentralizing integration ownership

In the strongly layered architecture described in "Chapter 3: The journey so far:

SOA, ESBs and APIs", technology islands such as integration had their own dedicated, and often centralized teams. Often referred to as the "ESB team" or the "SOA team", they owned the integration infrastructure, and the creation and maintenance of everything on it.
We could debate Conway's Law as to whether the architecture created the separate team or the other way around, but the more important point is that the technology restriction of needing a single integration infrastructure has been lifted.

We can now break integrations out into separate decoupled (containerized) pieces, each carrying all the dependencies they need, as demonstrated in Figure 9 below.

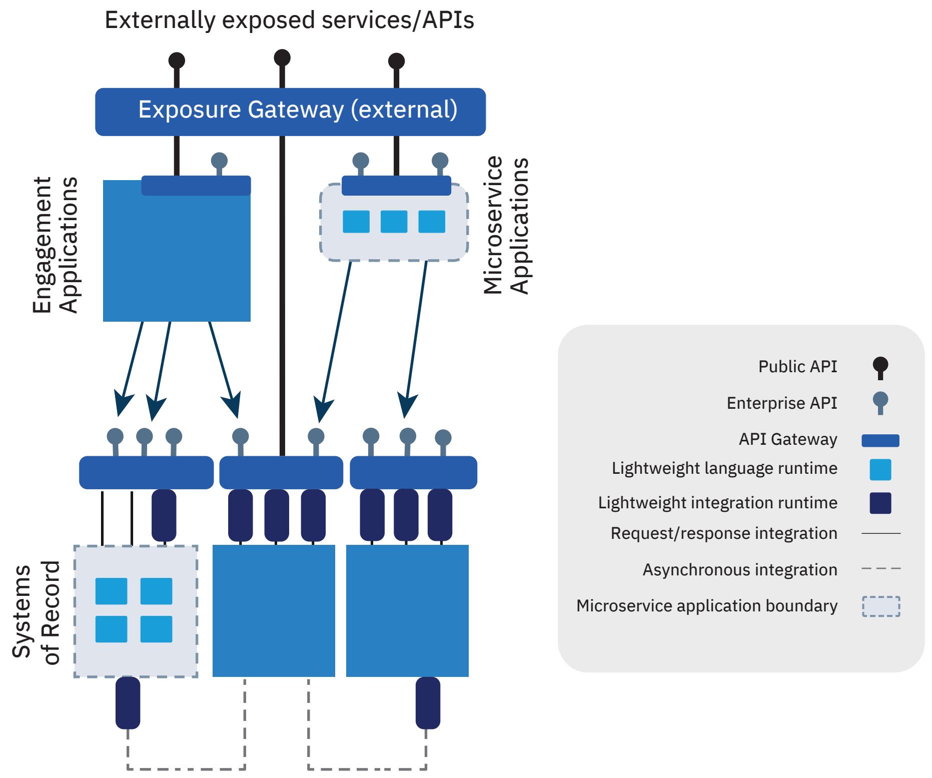

Externally exposed services/APIs
Exposure Gateway (external)
Engagement Applications
Microservice Applications
Systems of Record
Public API
Enterprise API
API Gateway
Lightweight language runtime
Lightweight integration runtime
Request/response integration
Asynchronous integration
Microservice application boundary

Figure 9: Decentralizing integration to the application teams

Technologically, there may be little difference between this diagram and the preceding fine-grained integration diagram in the previous chapter. All the same integrations are present, they're just in a different place on the diagram. What's changed is who owns the integration components. Could you have the application teams take on integration themselves? Could they own the creation and maintenance of the integrations that belong to their applications? This is feasible because not only have most integration runtimes become more lightweight, but they have also become significantly easier to use. You no longer need to be a deep integration specialist to use a good modern integration runtime. It's perfectly reasonable that an application developer could make good use of an integration runtime.

You'll notice we've also shown the decentralization of the gateways to denote that the administration of the API's exposure moves to the application teams as well.

There are many potential advantages to this decentralized integration approach:

- **Expertise:** A common challenge for separate SOA teams was that they didn't understand the applications they were offering through services. The application teams know the data structures of their own applications better than anyone.
- **Optimization:** Fewer teams will be involved in the end-to-end implementation of a solution, significantly reducing the cross-team chatter, project delivery timeframe, and inevitable waterfall development that typically occurs in these cases.
- **Empowerment:** Governance teams were viewed as bottle necks or checkpoints that had to be passed. There were artificial delays that were added to document, review then approve solutions.

The goal was to create consistency, the con is that to create that consistency took time. The fundamental question is “does the consistency justify the additional time?” In decentralization, the team is empowered to implement the governance policies that are appropriate to their scope.

Decentralized integration increases project expertise, focus and team empowerment.

Let’s just reinforce that point we made in the introduction of this chapter. While decentralization of integration offers potential unique benefits, especially in terms of overall agility, it is a significant departure from the way many organizations are structured today. The pros and cons need to be weighted carefully, and it may be that a blended approach where only some parts of the organization take on this approach is more achievable.

Does decentralized integration also mean decentralized infrastructure

To re-iterate, *decentralized integration is primarily an organizational change, not a technical one.* But does decentralized integration imply an infrastructure change? Possibly, but not necessarily.

The move toward decentralized ownership of integrations and their exposure does not necessarily imply a decentralized infrastructure. While each application team clearly could have its own gateways and container orchestration platforms, this is not a given. The important thing is that they can work autonomously.

API management is very commonly implemented in this way: with a shared infrastructure (an HA pair of gateways and a single installation of the API management components), but with each application team directly administering their own APIs as if they had their own individual infrastructure. The same can be done with the integration runtimes by having a centralized container orchestration platform on which they can be deployed but giving application teams the ability to deploy their own containers independently of other teams.

Benefits for cloud

It is worth noting that this decentralized approach is particularly powerful when moving to the cloud. Integration is already implemented in a cloud-friendly way and aligned with systems of record. Integrations relating to the application have been separated out from other unrelated integrations so they can move cleanly with the application. Furthermore, container-based infrastructures, if designed using cloud-ready principles and an infrastructure-as-code approach, are much more portable to cloud and make better use of cloud-based scaling and cost models. With the integration also owned by the application team, it can be effectively packaged as part of the application itself.

In short, decentralized integration significantly improves your cloud readiness.

We are now a very long way from the centralized ESB pattern—indeed, the term makes no sense in relation to this fully decentralized pattern—but we're still achieving the same intent of making application data and functions available for re-use by other applications across and even beyond the enterprise.

Traditional centralized technology-based organization

In the following Figure 10, we show how in a traditional SOA architecture, people were aligned based to their technology stack.

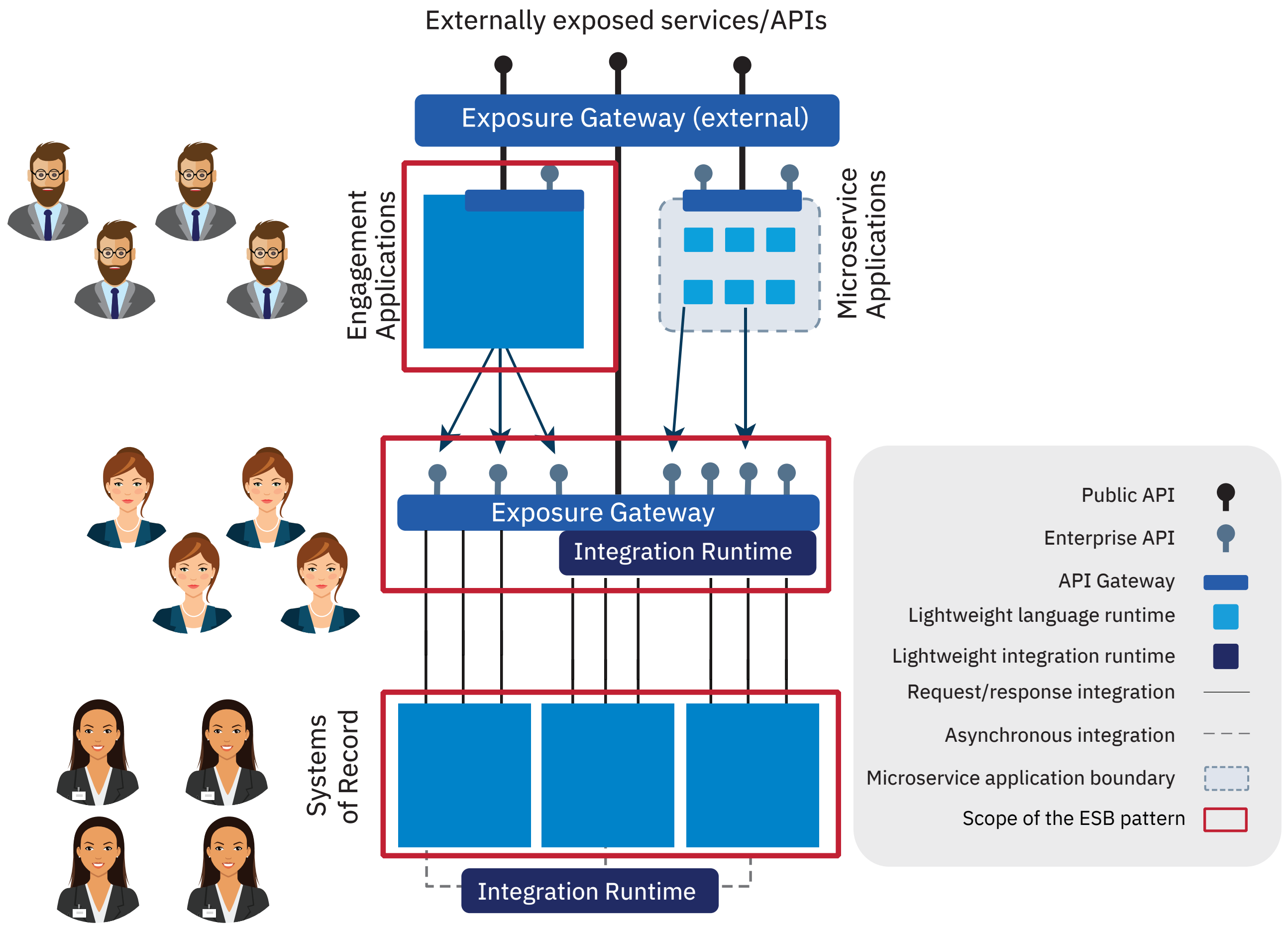

Externally exposed services/APIs
Exposure Gateway (external)
Engagement Applications
Microservice Applications
Exposure Gateway
Integration Runtime
Systems of Record
Integration Runtime
Public API
Enterprise API
API Gateway
Lightweight language runtime
Lightweight integration runtime
Request/response integration
Asynchronous integration
Microservice application boundary
Scope of the ESB pattern

Figure 10: Alignment of IT staff according to technology stack in an ESB environment.

A high level organizational chart would look something like this:

- A front-end team, which would be focused on end user's experience and focused on creating UIs.
- An ESB team, which would be focused on looking at existing assets that could be provided as enterprise assets. This team would also be focused on creating the services that would support the UIs from the front-end team.
- A back-end team, which would focus on the implementation of the enterprise assets surfaced through the ESB. There would be many teams here working on many different technologies. Some might be able to provide SOAP interfaces created in Java, some would provide COBOL copybooks delivered over MQ, yet others would create SOAP services exposed by the mainframe and so on.

This is an organizational structure with an enterprise focus which allows a company to rationalize its assets and enforce standards across a large variety of assets. The downside of this focus is that time to market for an individual project was compromised for the good of the enterprise.

A simple example of this would be a front-end team wanting to add a single new element to their screen. If that element doesn't exist on an existing SOAP service in the ESB then the ESB team would have to get engaged. Then, predictably, this would also impact the back-end team who would also have to make a change. Now, generally speaking, the code changes at each level were simple and straightforward, so that wasn't the problem.

The problem was allocating the time for developers and testers to work on it. The project managers would have to get involved to figure out who on their teams had capacity to add the new element, and how to schedule the push into the various environments. Now, if we scale this out we also have competing priorities. Each project and each new element would have to be vetted and prioritized, and all this is what took the time. So now we are in a situation where there is a lot of overhead, in terms of time, for a very simple and straightforward change.

The question is whether the benefits that we get by doing governance, and creating common interfaces is worth the price we pay for the operational challenges? In the modern digital world of fast-paced innovation we must think of a new way to enforce standards while allowing teams to reduce their time to market.

Moving to a decentralized, business-focused team structure

We're trying to reduce the time between the business ask and production implementation, knowing that we may rethink and reconsider how we implement the governance processes that were once in place. Let's now consider the concept of microservices and that we've broken our technical assets down into smaller pieces. If we don't consider reorganizing, we might actually make it worse! We'll introduce even more hand-offs as the lines of what is an application and who owns what begin to blur. We need to re-think how we align people to technical assets. In Figure 11, give you a preview of what that new alignment might look like.

Instead of people being centrally aligned to the area of the architecture they work on, they've been decentralized, and aligned to business domains. In the past, we had a front-end team, services teams, back-end teams and so on; now we have a number of business teams. For example, an Account team which works on anything related to accounts regardless whether or not the accounts involve a REST API, a microservice, or a user interface.

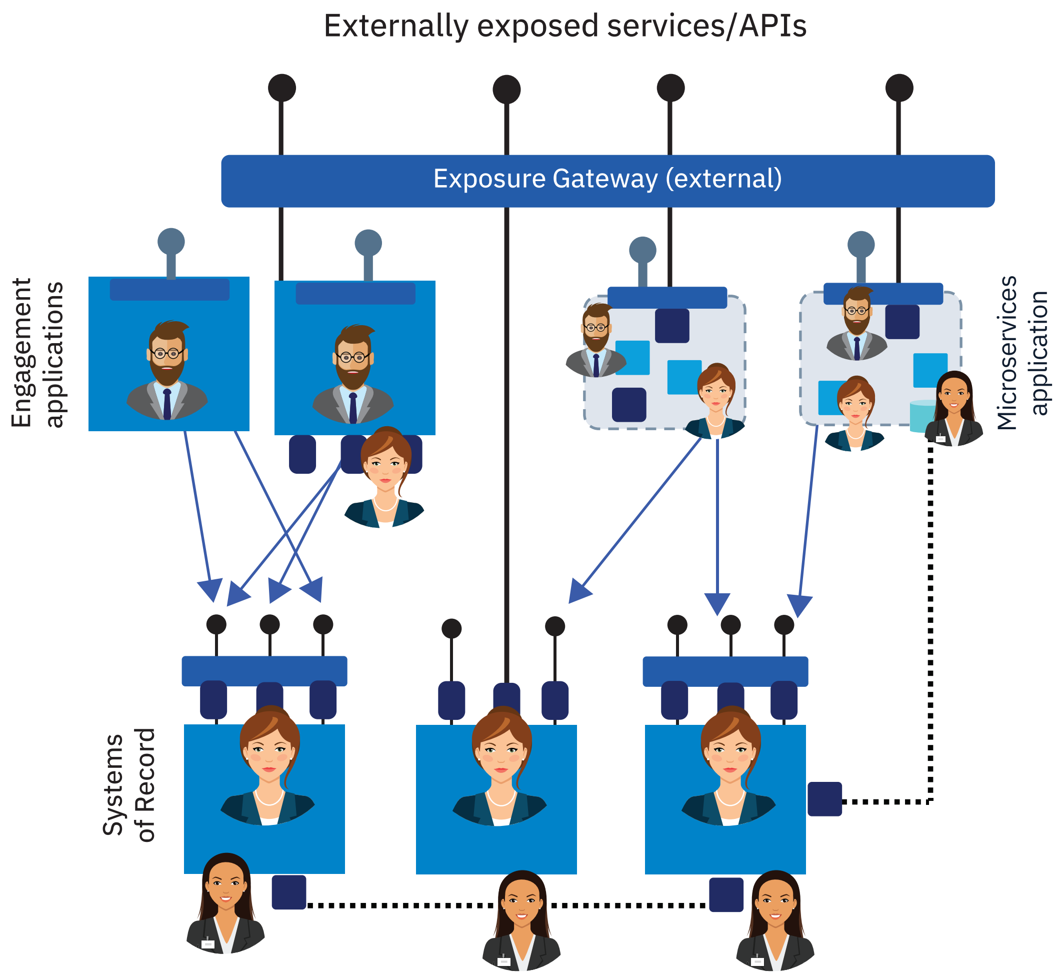

Externally exposed services/APIs
Exposure Gateway (external)
Engagement applications
Microservices application
Systems of Record

Figure 11: Decentralized IT staff structures.

The teams need to have cross-cutting skills since their goal is to deliver business results, not technology. To create that diverse skill set, it's natural to start by picking one person from the old ESB team, one person from the old front-end team, and another from the back-end team. It is very important to note that this does not need to be a big bang re-org across the entire enterprise, this can be done application by application, and piece by piece.

Big bangs generally lead to big disasters

The concept of "big bangs generally lead to big disasters" isn't only applicable to code or applications. It's applicable to organizational structure changes as well. An organization's landscape will be a complex heterogeneous blend of new and old. It may have a "move to cloud" strategy, yet it will also contain stable heritage assets. The organizational structure will continue to reflect that mixture. Few large enterprises will have the luxury of shifting entirely to a decentralized organizational structure, nor would they be wise to do so.

For example, if there is a stable application and there is nothing major on the road map for that application, it wouldn't make sense to decompose that application into microservices. Just as that wouldn't make sense, it also would not make sense to reorganize the team working on that application. Decentralization need only occur where the autonomy it brings is required by the organization, to enable rapid innovation in a particular area.

We certainly do not anticipate reorganization at a company level in its entirety overnight. The point here is more that as the architecture evolves, so should the team structure working on those applications, and indeed the integration between them. If the architecture for an application is not changing and is not foreseen to change there is no need reorganize the people working on that application.

Prioritizing Project Delivery First

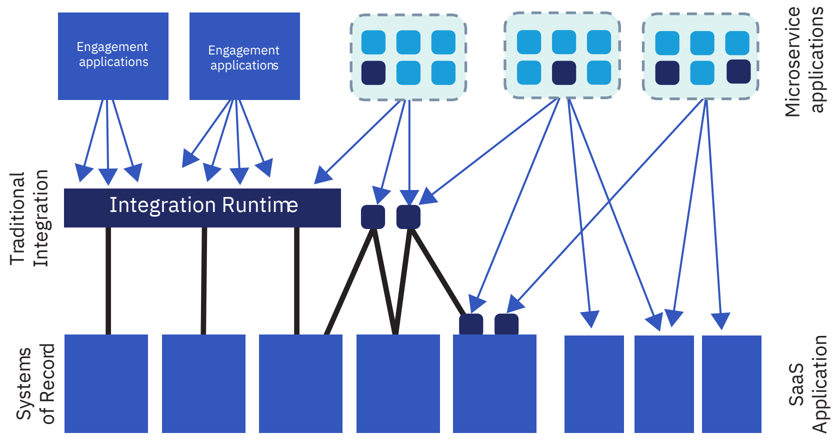
Now let’s consider what this change does to an individual and what they’re concerned about.

The first thing you’ll notice about the next diagram is that it shows both old and new architectural styles together. This is the reality for most organizations. There will be many existing systems that are older, more resistant to change, yet critical to the business. Whilst some of those may be partially or even completely re-engineered, or replaced, many will remain for a long time to come. In addition, there is a new wave of applications being built for agility and innovation using architectures such as microservices. There will be new cloud-based software-as-a-service applications being added to the mix too.

If we look into the concerns and motivations of the people involved, they fall into two very different groups, illustrated in Figure 12.

Re-use
Quality
Stability
Support
Monitoring
Governance
Performance
Fixed requirements

What’s its track recor Is the vendor trustworthy Will it serve me long term What do the analysts think of it Could I get sacked for a risky choice



Agility
Velocity
Autonomy Freemium
Cloud native
Vendor agnostic
Developer is king
Rapid prototyping
Short learning curve

Can I start small Can it help me today What do my peers think of it Does it have an active community Are my skills relevant to my peers

Figure 12: Traditional developers versus a gile teams

A developer of traditional applications cares about stability and generating code for re-use and doing a large amount of up-front due diligence. The agile teams on the other hand have shifted to a delivery focus. Now, instead of thinking about the integrity of the enterprise architecture first and being willing to compromise on the individual delivery timelines, they’re now thinking about delivery first and willing to compromise on consistency.

Agile teams are more concerned with the project delivery than they are with the enterprise architecture integrity.

Let's view these two conflicting priorities as two ends of a pendulum. There are negatives at the extreme end on both sides. On one side, we have analysis paralysis where all we're doing is talking and thinking about what we should be doing, on the other side we have the wild-wild-west were all we're doing is blindly writing code with no direction or thought towards the longer-term picture. Neither side is correct, and both have grave consequences if allowed to slip too far to one extreme or the other. The question still remains: "If I've broken my teams into business domains and they're enabled and focused on delivery, how do I get some level of consistency across all the teams? How do I prevent duplicate effort? How do I gain some semblance of consistency and control while still enabling speed to production?"

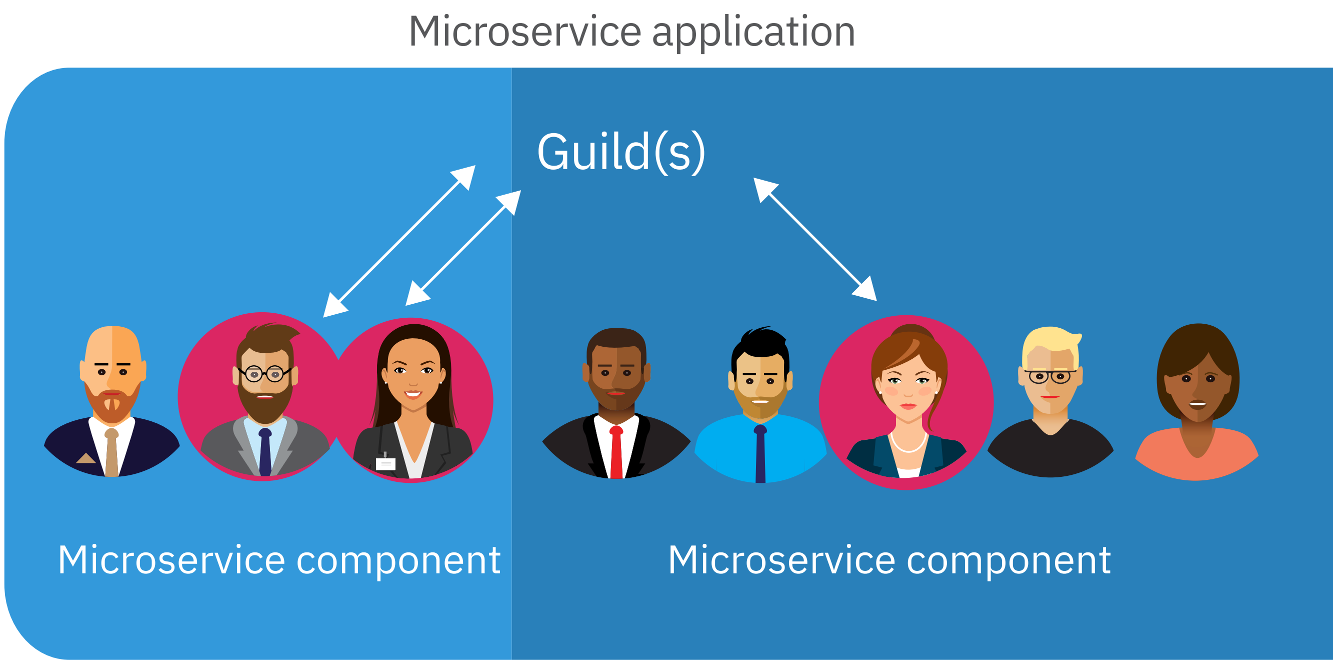


Figure 13: Practicing architects play a dual role as individual contributors and guild members.

Evolving the role of the Architect

The answer is to also consider the architecture role. In the SOA model the architecture team would sit in an ivory tower and make decisions. In the new world, the architects have an evolved role--practicing architects. An example is depicted in Figure 13.

Here we have many teams and some of the members of those teams are playing a dual role. On one side they are expected to be an individual contributor on the team, and on the other side they sit on a committee (or guild) that rationalizes what everyone is working on. They are creating common best practices from their work on the ground. They are creating shared frameworks, and sharing their experiences so that other teams don't blunder into traps they've already encountered. In the SOA world, it was the goal to stop duplication/enforce standards before development even started. In this model the teams are empowered, and the committee or guild's responsibility is to raise/address and fix cross cutting concerns at the time of application development.

If there is a downside to decentralization, it may be the question of how to govern the multitude of different ways that each application team might use the technology – essentially encouraging standard patterns of use and best practices. Autonomy can lead to divergence.

If every application team creates APIs in their own style and convention, it can become complex for consumers who want to re-use those APIs. With SOA, attempts were made to create rigid standards for every aspect of how the SOAP protocol would be used, which inevitably made them harder to understand and reduced adoption. With RESTful APIs, it is more common to see convergence on conventions rather than hard standards. Either way, the need is clear: Even in decentralized environments, you still need to find ways to ensure an appropriate level of commonality across the enterprise. Of course, if you are already exploring a microservices-based approach elsewhere in your enterprise, then you will be familiar with the challenges of autonomy.

The practicing architect is now responsible for execution of the individual team mission as well as the related governance requirements that cut across the organization.

Therefore, the practicing architect is now responsible for knowing and understanding what the committee has agreed to, encouraging their team to follow the governance guidelines, bringing up cross-cutting concerns that their team has identified, and sharing what they're working on. This role also has the need to be an individual contributor on one of the teams so that they feel the pain, or benefit, of the decisions made by the committee.

Enforcing governance in a decentralized structure

With the concept of decentralization comes a natural skepticism over whether the committee or guild's influence will be persuasive enough to enforce the standards they've agreed to. Embedding our "practicing architect" into the team may not be enough.

Let's consider how the traditional governance cycle often occurs. It often involves the application team working through complex standards documents, and having meetings with the governance board prior to the intended implementation of the application to establish agreement. Then the application team would proceed to development activities, normally beyond the eyes of the governance team. On or near completion, and close to the agreed production date, a governance review would occur.

Inevitably the proposed project architecture and the actual resultant project architecture will be different, and at times, radically different. Where the architecture review board had an objection, there would almost certainly not be time to resolve it. With the exception of extreme issues (such as a critical security flaw), the production date typically goes ahead, and the technical debt is added to an ever-growing backlog.

Clearly the shift we've discussed of placing practicing architects in the teams encourages alignment. However, the architect is now under project delivery pressure which may mean they fall into the same trap as the teams originally did, sacrificing alignment to hit deadlines. What more can we do, via the practicing architect role, to encourage enforcement of standards?

The key ingredient for success in modern agile development environment is automation: automated build pipelines, automated testing, automated deployment and more. The practicing architect needs to be actively involved in ways to automate the governance.

This could be anything from automated code review, to templates for build pipelines, to standard Helm charts to ensure the target deployment topologies are homogeneous even though they are independent. In short, the focus is on enforcement of standards through frameworks, templates and automation, rather than through complex documents, and review processes. While this idea of getting the technology to enforce the standards is far from new, the proliferation of open standards in the DevOps tool chain and cloud platforms in general is making it much more achievable.

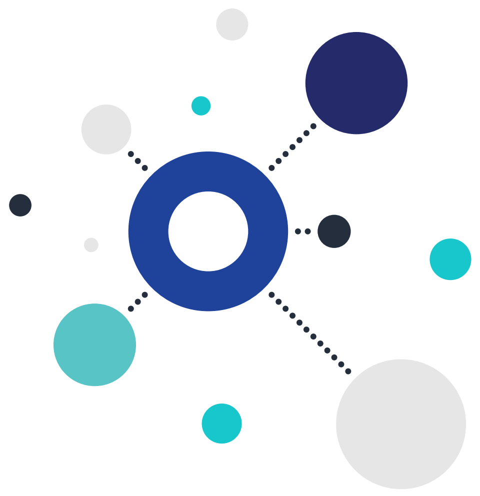
Let’s start with an example: say that you have microservices components that issue HTTP requests. For every HTTP request, you would like to log in a common format how long that HTTP transaction took as well as the HTTP response code. Now, if every microservice did this differently, there wouldn’t be a unified way of looking at all traffic. Another role of the practicing architect is to build helper artifacts that would then be used by the microservices. In this way, instead of the governance process being a gate, it is an accelerator through the architects being embedded in the teams, working on code alongside of them. Now the governance cycle is being done with the teams, and instead of reviewing documents, the code is the document and the checkpoint is to make sure that the common code is being used.

Another dimension to note is that not all teams are created equally. Some teams are cranking out code like a factory, others are thinking ahead to upcoming challenges, and some teams are a mix of the two. An advanced team that succeeds in finding a way to automate a particular governance challenge will be much more successful evangelists for that mechanism than any attempt for it to be created by a separate governance team.

As we are discussing the technical architect it may seem that too much is being put on their shoulders. They are responsible for application delivery, they are responsible to be a part of the committee discussed in the previous section, and now we are adding on an additional element of writing common code that is to be used by other application development teams. Is it too much?

A common way to offload some of that work is to create a dedicated team that is under the direction of the practicing architect who is writing and testing this code. The authoring of the code isn’t a huge challenge, but the testing of that common code is. The reason for placing a high value on testing is because of the potential impact to break or introduce bugs into all the applications that use that code. For this reason, extra due diligence and care must be taken, justifying the investment in the additional resource allocation.

Clearly our aim should be to ensure that general developers in the application teams can focus on writing code that delivers business value. With the architects writing or overseeing common components which naturally enforce the governance concerns, the application teams can spend more of their time on value, and less in governance sessions. Governance based on complex documentation and heavy review procedures are rarely adhered to consistently, whereas inline tooling based standardization happens more naturally.



How can we have multi-skilled developers?

The next and very critical person to consider is the developer. Developers are now to be expected and encouraged to be a full stack developer and solve the business problem with whatever technology is required. This puts an incredible strain on each individual developer in terms of the skills that they must acquire. It's not possible for the developer to know the deep ins and outs of every aspect of each technology, so something has to give. As we'll see, what gives is the infrastructure learning curve – we are finding better and better ways to make infrastructural concerns look the same from one product to another.

In the pre-cloud days, developers had to learn multiple aspects of each technology as categorized in Figure 14.

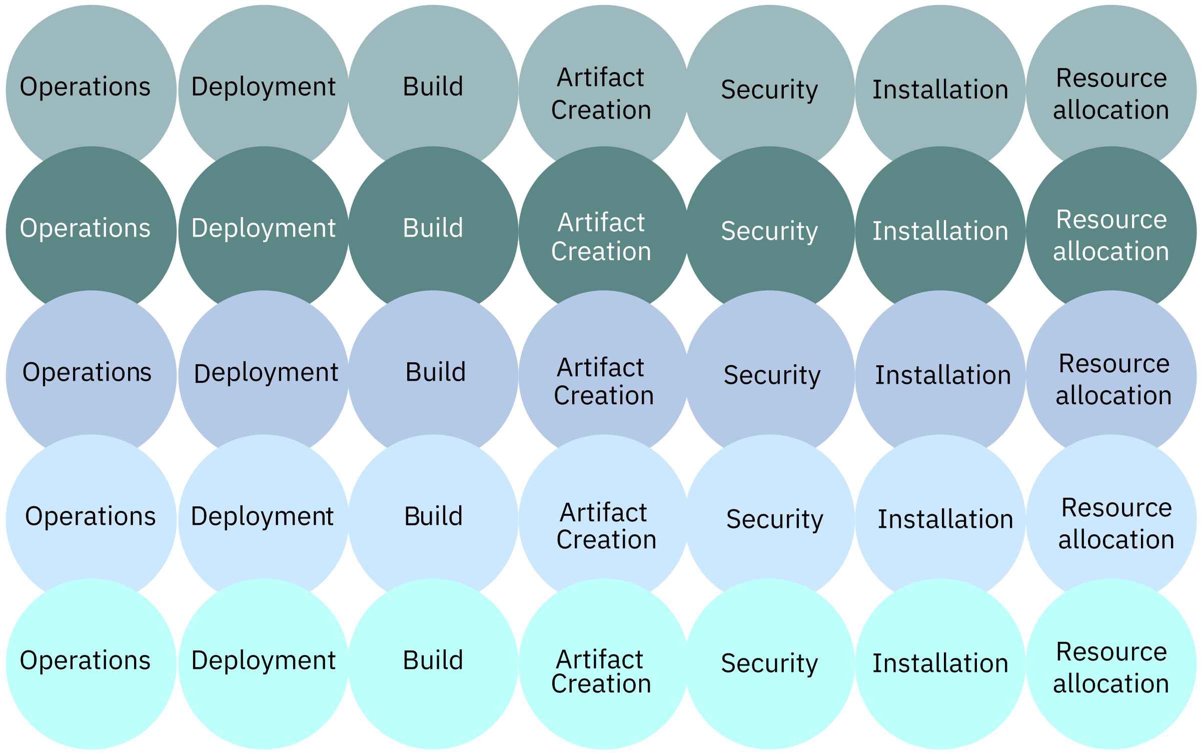


Figure 14: Required pre-cloud technology skills.

Decentralization allows developers to focus on what their team is responsible for; delivering business results by creating artifacts.

Each column represents a technology and each row represents an area that the developer had to know and care about, and understand the implications of their code on. They had to know individually for each technology how to install, how much resources it would need allocated to it, how to cater for high availability, scaling and security. How to create the artifacts, how to compile and build them, where to store them, how to deploy them, and how to monitor them at runtime. All this unique and specific to each technology. It is no wonder that we had technology specific teams!

However, the common capabilities and frameworks of typical cloud platforms now attempt to take care of many of those concerns in a standardized way. They allow the developer to focus on what their team is responsible for, delivering business results by creating artifacts! Figure 15 shows how decentralization removes the ‘white noise’.

The grey area represents areas that still need to be addressed but are now no longer at the front of the developer’s mind. Standardized technology such as (Docker) containers, and orchestration frameworks such as Kubernetes, and routing frameworks such as Istio, enable management of runtimes in terms of scaling, high availability, deployment and so on. Furthermore, standardization in the way products present themselves via command line interfaces, APIs, and simple file system-based install and deployment mean that standard tools can be used to install, build and deploy, too.

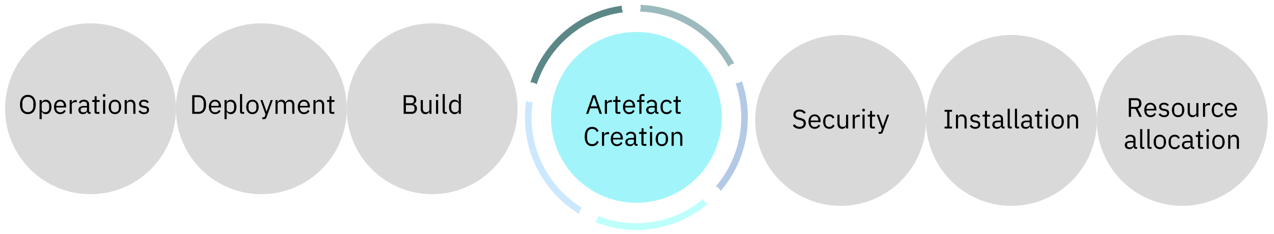


Figure 15: Required pre-cloud technology skills.

One day, in an ideal world, the only unique thing about using a technology will be the creation of the artifact such as the code, or in the case of integration, the mediation flows and data maps. Everything else will come from the environment. We’ll discuss this infrastructural change in more depth in the next chapter.

Conclusions on decentralized integration ownership

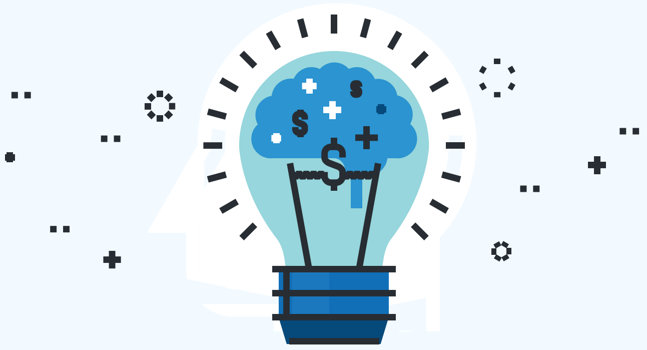
Of course, decentralization isn't right for every situation. It may work for some organizations, or for some parts of some organizations but not for others. Application teams for older applications may not have the right skill sets to take on the integration work. It may be that integration specialists need to be seeded into their team. This approach is a tool for potentially creating greater agility for change and scaling, but what if the application has been largely frozen for some time?

At the end of the day, some organizations will find it more manageable to retain a more centralized integration team. The approach should be applied where the benefits are needed most. That said, this style of decentralized integration is what many organizations and indeed application teams have always wanted to do, but they may have had to overcome certain technological barriers first.

The core concept is to focus on delivering business value and a shift from a focus on the enterprise to a focus on the developer. This concept has in part manifested itself by the movement from centralized teams into more business specific ones, but also by more subtle changes such as the role of a practicing architect.

This concept is also rooted in actual technology improvements that are taking concerns away from the developer and doing those uniformly through the facilities of the cloud platform.

As ever, we can refer right back to Conway's Law (circa 1967) - if we're changing the way we architect systems and we want it to stick, we also need to change the organizational structure.



Lessons Learned

A real-life scenario

An organization who committed to decentralization was working with a microservices architecture that had now been widely adopted, and many small, independent assets were created at a rapid pace. In addition to that, the infrastructure had migrated over to a Docker-based environment. The organization didn't believe they needed to align their developers with specific technical assets.

The original thought was that any team could work on any technical component. If the feature required a team to add an element onto an existing screen, that team was empowered and had free range to modify whatever assets were needed to to accomplish the business goal. There was a level of coordination that occurred before the feature was worked on so that no two teams would be working on the same code at the same time. This avoided the need for merging of code.

In the beginning, for the first 4-5 releases, this worked out beautifully. Teams could work independently and could move quickly. However, over time problems started to arise.

The problem

The main problem was lack of end state vision. Because each piece of work was taken independently teams often did the minimum amount of work to accomplish the business objective. The main motivators for each team were risk avoidance and drive to meet project deadlines – and a desire not to break any existing functionality. Since each team had little experience with the code they needed to change, they began making tactical decisions to lower risk.

Developers were afraid to break currently working functionality. As they began new work, they would work around code that was authored from another team. Therefore, all new code was appended to existing code. The microservices continued growing and growing over time, which then resulted in the microservices not being so micro.

This lead to technical debt piling up. This technical debt was not apparent over the first few releases, but then, 5 or 6 releases in, this became a real problem. The next release required the investment of unravelling past tactical decisions. Over time the re-hashing of previously made decisions outweighed the agility that this organization structure had originally produced.

The solution

The solution was to align teams to microservices components, and create clear delineation of responsibilities. These needed to be done through a rational approach. The first step was to break down the entire solution into bounded contexts, then assign teams ownership over those bounded context. A bounded context is simply a business objective and a grouping of business functions. An individual team could own many microservices components, however those assets all had to be aligned to the same business objective. Clear lines of ownership and responsibility meant that the team thought more strategically about code modifications. The gravity of creating good regression tests was now much more important since each team knew they would have to live with their past decisions.

Importantly, another dimension of these new ownership lines meant less handoffs between teams to accomplish a business objective. One team would own the business function from start to finish - they would modify the front-end code, the integration layer and the back-end code, including the storage. This grouping of assets is clearly defined in microservices architecture, and that principle should also carry through to organization structures to reduce the handoffs between teams and increase operational efficiency.

Chapter 6: Aspect 3: Cloud native integration infrastructure

If we are to be truly affective in transitioning to an agile integration architecture, we will need to do more than simply break out the integrations into separate containers. We also need to apply a cloud native - “cattle not pets” - approach to the design and configuration of our integrations.

As a result of moving to a fully cloud native approach, integration then becomes just another option in the toolbox of lightweight runtimes available to people building microservices based applications. Instead of just using integration to connect applications together, it can now also be used within applications where a component performs an integration centric task.

Cattle not pets

Let take a brief look at where that concept came from before we discuss how to apply it in the integration space.

In a time when servers took weeks to provision and minutes to start, it was fashionable to boast about how long you could keep your servers running without failure. Hardware was expensive, and the more applications you could pack onto a server, the lower your running costs were. High availability (HA) was handled by using pairs of servers, and scaling was vertical by adding more cores to a machine. Each server was unique, precious, and treated, well, like a pet.

Times have changed. Hardware is virtualized. Also, with container technologies, such as Docker, you can reduce the surrounding operating system to a minimum so that you can start an isolated process in seconds at most. Using cloud-based infrastructure, scaling can be horizontal, adding and removing servers or containers at will, and adopting a usage-based pricing model. With that freedom, you can now deploy thin slivers of application logic on minimalist runtimes into lightweight independent containers. Running significantly more than just a pair of containers is common and limits the effects of one container going down. By using container orchestration frameworks, such as Kubernetes, you can introduce or dispose of containers rapidly to scale workloads up and down. These containers are treated more like a herd of cattle.

Using cloud-based infrastructure provides freedom to deploy thin slivers of application logic on minimalist runtimes into lightweight independent containers.

Integration pets: The traditional approach

Let’s examine what the common “pets” model looks like. In the analogy, if you view a server (or a pair of servers that attempt to appear as a single unit) as indispensable, it is a pet. In the context of integration, this concept is similar to the centralized integration topologies that the traditional approach has used to solve enterprise application integration (EAI) and service-oriented architecture use cases.

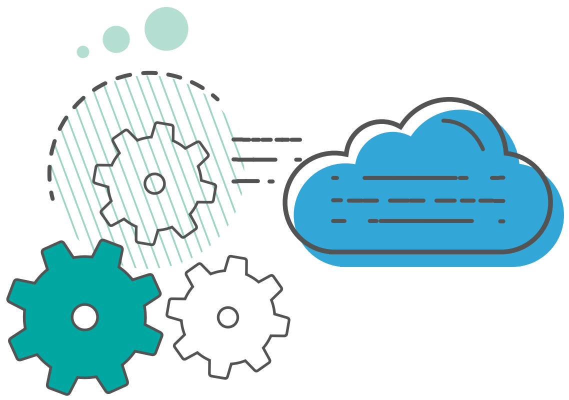


Table 1. Characteristics of pets

General characteristics of pets	How they are applied to a centralized or traditional integration context
Manually built	Integration hubs are often built only once in the initial infrastructure stage. Scripts help with consistency across environments but are mostly run manually.
Managed	The hub and its components are directly and individually monitored during operation with a role-based access control to allow administrative access to different groups of users.
Hand fed	The hub is nurtured over time, for example, by introducing new integration applications, and changes to OS and software maintenance levels. As part of this process, new options and parameters are applied, changing the overall configuration of the hub. Thus, even if the server started out being based on a defined pattern, gradually the running instance becomes more bespoke with each change in comparison to the original installation.
Server pairs	Typically pairs of nodes provide HA. Great care is taken to keep these pairs up and running and to back up the evolving configuration. Scalability is coarse-grained and achieved by creating more pairs or adding resources so that existing pairs can support more workloads.

Integration cattle: An alternative lightweight approach

Simplistically, this shift means breaking up the more centralized ESB runtime into multiple separate and highly decoupled run times. However, the change involves more than just breaking out the integrations into containers. A cattle-based approach must exhibit many, if not all, of the characteristics in Table 2.

Adopting such an approach then impacts the ways in which your DevOps teams will interact with the environment and the solution overall. These will be consistent across any solution that exists in a container-based architecture, which will help create efficiencies as more solutions are moved to lightweight architectures.

Table 2. Characteristics of cattle

Characteristics of cattle	How they are applied to a agile integration architecture context
Elastic scalability	Integrations are scaled horizontally and allocated on-demand in a cloud-like infrastructure.
Disposable and re-creatable	Using lightweight container technology encourages changes to be made by redeploying amended images rather than by nurturing a running server.
Starts and stops in seconds	Integrations are run and deployed as more fine-grained entities and, therefore, take less time to start.
Minimal interdependencies	Unrelated integrations are not grouped. Functional and operational characteristics create colocation and grouping
Infrastructure as code	Resources and code are declared and deployed together.

- **Maintenance:** Integration servers are not administered live. If you want to make any adjustments such as change an integration, add a new one, change property values, add product fixpacks and so on, this is done by creating a new container image, starting up a new instance based on it, and shutting down the current container.

  Why? Any live changes to a running sever make it different from the image it was built from – it changes its runtime state. This would then mean that the container orchestration engine cannot re-create containers at will for failover and scaling.

- **Monitoring:** Monitoring isn't done via connecting to a live running server. Instead, the servers report what's going on inside them via logging, which is aggregated by the platform to provide a monitoring view.

  Why? Direct monitoring techniques would not be able to keep up with the constantly changing number of containers, nor would it be appropriate to expect every container to accept monitoring requests alongside its "day job". Note: There are some exceptions such as a simple health check, which is used by the container orchestration platform to determine if the server is functioning correctly and replace it if required.

- **Affinity:** Integration servers cannot make any assumptions about how many other replicas are running or where they are. This means careful consideration needs to be paid to anything that implies any kind of affinity, or selective caching of any data.

  Why? In a word, scalability. The container orchestration platform must be able to add or remove instances at will. If state is held for any reason, it will not be retained during orchestration.

Adopting a "cattle approach" impacts the ways in which your DevOps teams will interact with the environment and the solution overall, create increasing efficiencies as more solutions are moved to lightweight architectures.

There are plenty of additional considerations we could discuss, but the overall point is clear: we need to think very differently about how we design, build, and deploy if we are to reap the benefits of greater development agility, elastic scaling, and powerful resilience models.

What's so different with cattle

How do we know if we're doing it right? Are we really creating replaceable, scalable cattle, or do we still have heavily nurtured pets?

There are many elements to what constitutes an environment made from cattle rather than pets. One important litmus test that we'll discuss here revolves around the question "What is a part of your build package for each new version of a component?". Take a look at the two images in Figure 16.

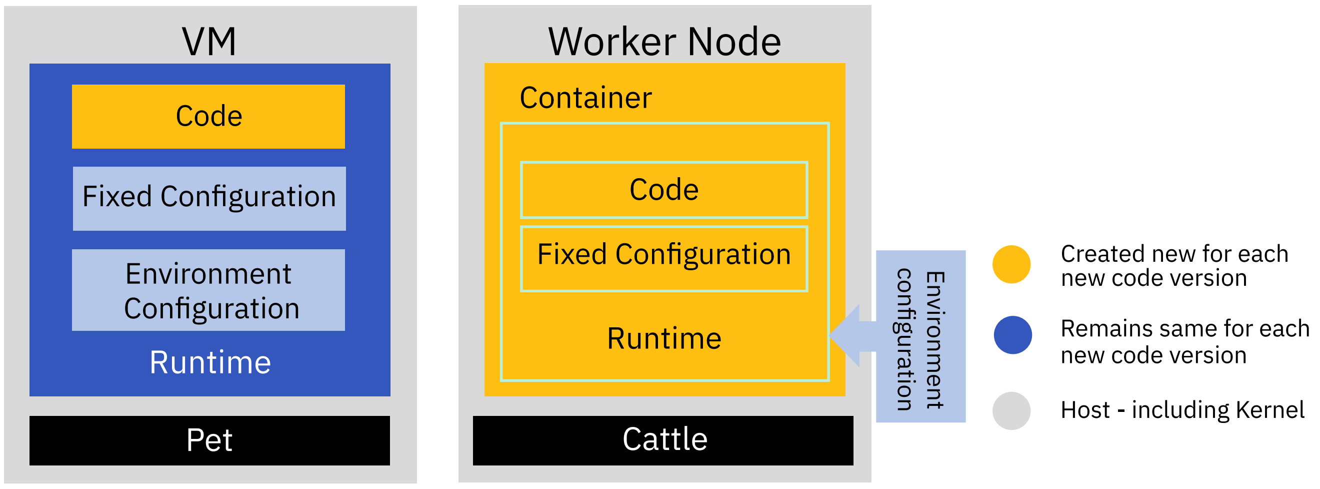

VM
Code
Fixed Configuration
Environment Configuration
Runtime
Pet
Worker Node
Container
Code
Fixed Configuration
Runtime
Cattle
Environment configuration
Created new for each new code version
Remains same for each new code version
Host - including Kernel

Figure 16: Pets versus Cattle

Let's start by defining what is meant by the text in the diagram

- **Code:** This is the code that you author and deploy as a unit. In a Java world this would be your JAR/EAR/WAR file. In a Node.js world, this would be the js files. In an IBM Integration Bus (IIB) world, this would be your BAR file.
- **Fixed Configuration:** These would be the dependencies that your code relies on. If your code is making an HTTP call, this would be the HTTP package that you're using. If you're using a Database connection, this would be the ODBC or JDBC classes.
- **Environment Configuration:** This would be the endpoints that are expected to change environment by environment. An example here would be if you're integrating with something over HTTP, this configuration would be the HTTP endpoint. If you're connecting to a Database then this would be the host, port, username and password.
- **Runtime:** This is what is running your code. It could either be your node runtime, java JRE, Liberty server, IIB server, MQ server, etc. It is the runtime that interprets and runs your coded artifacts.

An important question to ask when you release a new version is what is the scope of the build package. If it is code, and only code, then you are treating your server like a pet. This implies that version upgrades and patches would be done at a separate time and through a separate mechanism, leaving you unable to guarantee the consistency of the delivered artifacts.

This also implies that you couldn't spin up a new server quickly enough to meet the demands of elastic scaling.

If the answer to the litmus test question was everything including code, fixed configuration, runtime and environment configuration, then you are more than likely treating your servers as cattle.

This removes the chance that dev and production react differently due to some difference in server configuration since the server configuration is packaged alongside the code -"infrastructure as code".

Pros and cons

While we're clearly encouraging you to consider the benefits of moving to a more cattle-like approach, it's only fair to recognize that more traditional pet-like approach also has benefits that might be more challenging to achieve with cattle. For a quick comparison, see Figure 17, which shows some of the characteristics that vary between cattle and pets.

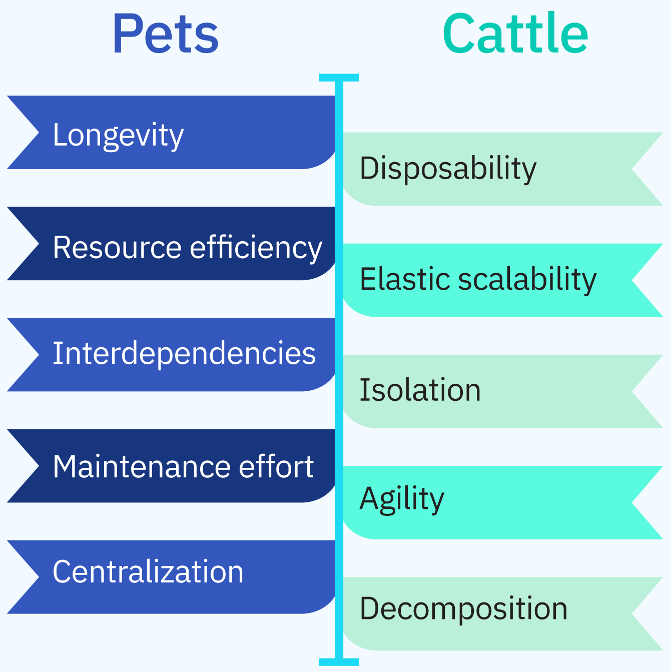


Figure 17: Characteristics of Pets and Cattle

Integration scenarios vary in the characteristics that they need. With modern approaches to more lightweight runtimes and containers, you have the opportunity to stand up each integration in the way that is most suited to it. You do not need to assume that just because a cattle approach suits many integrations, it will suit all of them. For example, existing integrations that rarely if ever need to be changed, and have predictable load may not gain any immediate benefit from the cattle approach. Conversely, new integrations likely to undergo regular amendments with as yet unknown loads will benefit significantly. You can use both approaches and even add hybrid options as required.

Application and integration handled by the same team

Once the application development group has taken on the integration, there's an elephant in the room: At what point are they doing integration, as opposed to application development?

For good reason, integration teams were often told they should only do integration logic, not application logic. This was to avoid spreading business logic across different teams and components throughout the enterprise. This deep divide between teams doing "application" and "integration" constantly dogged SOA, resulting in a cascade of waterfall-style requirements between the teams that slowed projects down.

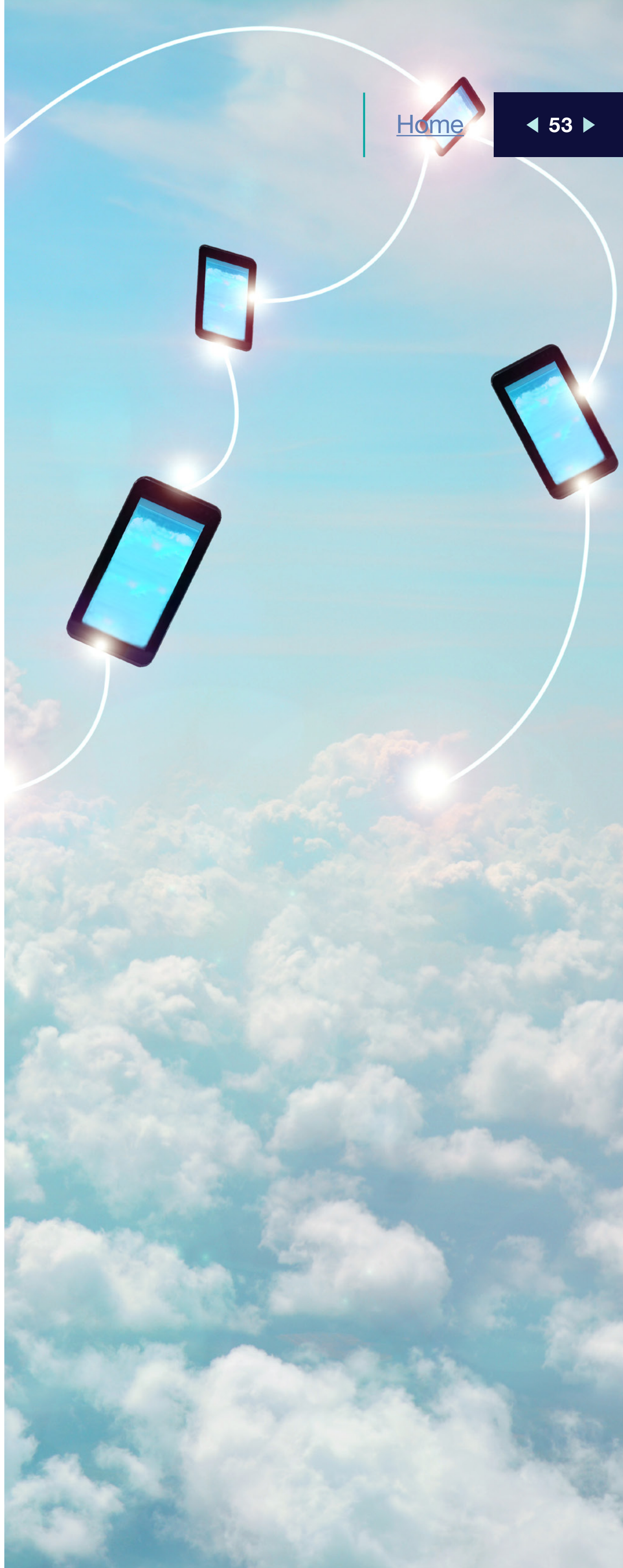
Now let's be clear here, the fundamental premise of separating integration from application is still important, but we no longer need to go to the extremes of having it done by separate teams – that was just enforced on us by the technology of the time.

What if, as in the previous section on decentralization, we moved the integration responsibility into the application team, and that team happened to be building their application using a microservices architecture? One of the key benefits of microservice architecture is that you can use multiple different runtimes, each best suited to the job in hand. For example, one runtime might be focused on the user interface and perhaps be based on Node.js and a number of UI libraries. Another runtime might be more focused on a particular need of the solution, such as a rules engine or machine learning. Of course, all applications need to get data in and out, so surely we would expect to also see an integration runtime too.

Integration technology in a microservices architecture can be a high-productivity part of any application.

It is common to find microservices components in an application whose responsibilities are primarily focused around integration. For example, what if all a microservice component did was to provide an API that performed a few invocations to other systems, collated and merged the results, and responded to the caller? That sounds a lot like something an integration tool would be good at. A simple graphical flow—one that showed which systems you're calling, allowed you to easily find where the data items are merged, and provided a visual representation of the mapping—would be much easier to maintain in the future than hundreds of lines of code.

Let's look at another example. There's a resurgence of interest in messaging in the microservices world, through the popularity of patterns such as event-sourced applications and the use of eventual-consistency techniques. So, you'll probably find plenty of microservice components that do little more than take messages from a queue or topic, do a little translation, and then push the result into a data store. However, they may require a surprisingly large number of lines of code to accomplish. An integration runtime could perform that with easily configurable connectors and graphical data mapping, so you don't have to understand the specifics of the messaging and data store interfaces, as depicted in Figure 18.



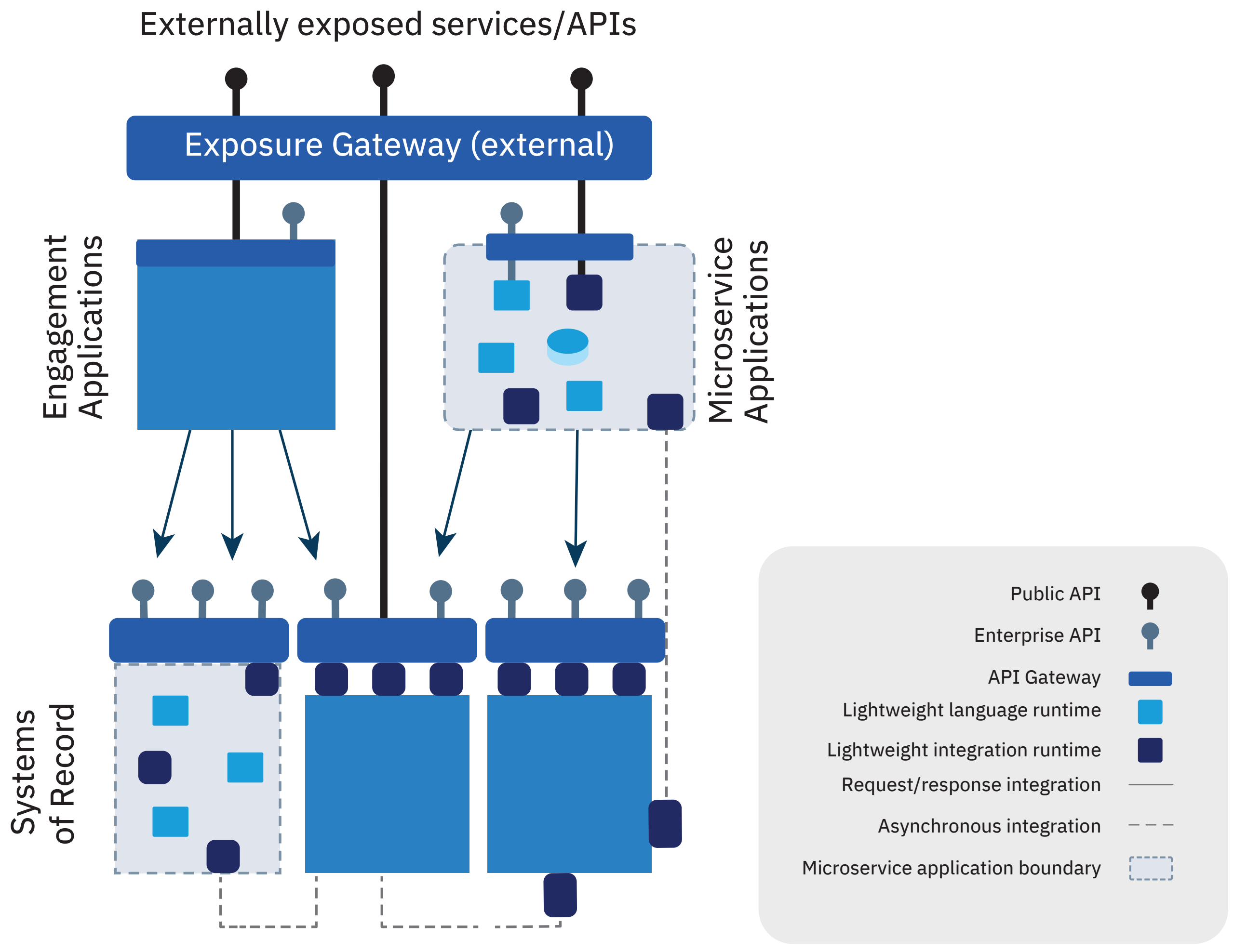

Externally exposed services/APIs
Exposure Gateway (external)
Engagement Applications
Microservice Applications
Systems of Record
Public API
Enterprise API
API Gateway
Lightweight language runtime
Lightweight integration runtime
Request/response integration
Asynchronous integration
Microservice application boundary

Figure 18: Using a lightweight integration runtime as a component within a microservices application

As you saw in previous sections, the integration runtime is now a truly lightweight component that can be run in a cloud-native style. Therefore, it can easily be included within microservices applications, rather than just being used to integrate between them.

When discussing this approach, an inevitable question is Am I introducing an ESB into a microservices application? It is an understandable concern, but it is incorrect, and it's extremely important to tackle this concern head on. As you may recall from the earlier definitions, an integration runtime is not an ESB. That is just one of the architectural patterns the integration runtime can be a part of.

ESB is the heavily centralized, enterprise-scope architectural pattern discussed earlier in Chapter 3. Using a modern lightweight integration runtime to implement integration-related aspects of an application, deploying each integration independently in a separate component is very different indeed from the centralized ESB pattern. So the answer is no, by using a lightweight integration runtime to containerize discrete integrations you are most certainly not re-creating the centralized ESB pattern within your microservices application.

One of the key benefits of microservices architecture is that you are no longer restricted to one language or runtime, which means you can have a **polyglot runtime**—a collection of different runtimes, each suited to different purposes. You can introduce integration as just another of the runtime options for your microservices applications. Whenever you need to build a microservices component that's integration centric, you would then expect to use an integration runtime.

Traditionally, integration runtimes have been mostly used for integration between separate applications—and they will certainly continue to perform that role—but here we are discussing its use as a component within an application.

In the past, it would have been difficult for application developers to take on integration since the integration tooling wasn't part of the application developer's toolbox. Deep skills were often required in the integration product and in associated integration patterns. Today, with the advances in simplicity of integration runtimes and tooling, there is no longer a need for a separate dedicated team to implement and operate them. Integrations are vastly easier to create and maintain.

In a world where applications are now composed of many fine-grained components that can be based on a polyglot of different runtimes, we now have the opportunity to use the right runtime for each task at hand. Where integration-like requirements are present, we can choose to use an integration runtime.

Common infrastructure enabling multi-skilled development

What is it exactly that has made it possible for microservice application teams to work with multiple different languages and runtimes within their solution. Certainly, in part it comes down to the fact that languages have become more expressive – you can achieve more, with less lines of code – and tooling has become easier to learn and more powerful. However, there's another key reason that is directly related to what cloud-native brings to the table. The runtimes share a common infrastructure not just at the operating system level, but in many other dimensions.

Historically, each runtime type came with its own proprietary mechanisms for high-availability, scaling, deployment, monitoring and other system administration tasks.

Figure 19 demonstrates the difference between traditional and cloud native infrastructures.

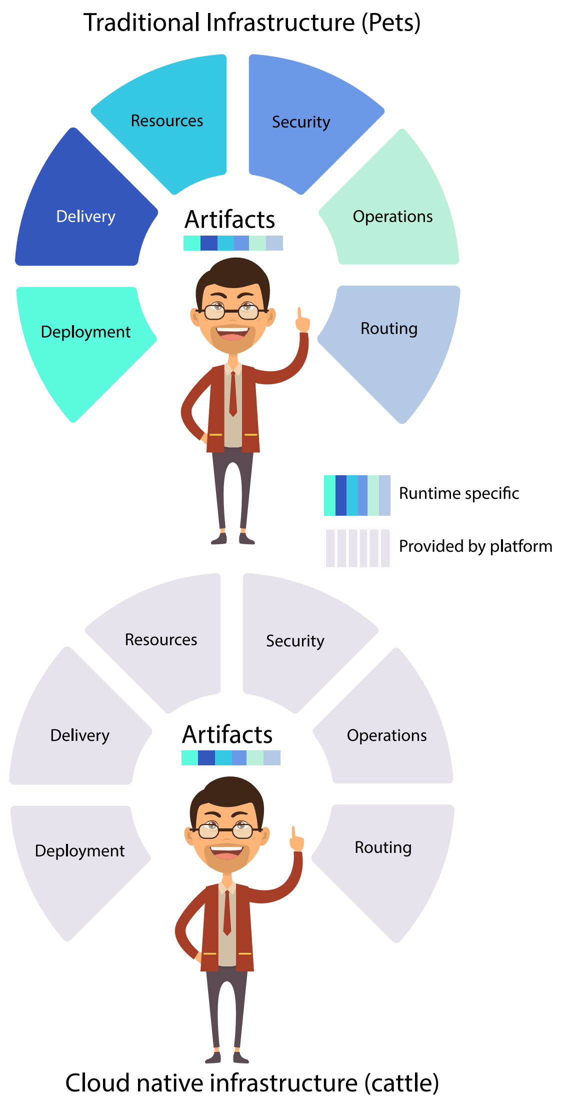


Figure 19: Traditional infrastructure with every capability tied to a specific runtime, and a cloud native nfrastructure with almost all capabilities provided by the platform.

Modern lightweight runtimes are designed to leverage many if not all of those capabilities from the platform in which they sit. Cloud native platforms such as Kubernetes combined with suitable runtime frameworks enable a lightweight runtime to be made highly available, scaled, monitored and more in a single standardized way rather than in a different way for each runtime.

Essentially the team only needs to gain one set of infrastructure skills and they can then look after the polyglot of runtimes in the application. This standardization extends into common source code repositories such as GitHub and build tools such as Jenkins. It also increases the consistency of deployment as you are propagating pre-built images that include all dependencies out to the environments. Finally, it simplifies install by simply layering files onto the file system.

Ideally, the only new skills you need to pick up to use another runtime is how to build its artifacts, whether that be writing code for a language runtime, or building mediation flows for an integration engine. Everything else is done the same way across all runtimes.

Once again, this brings the freedom to choose the best runtime for the task at hand. Based on the information above, it is clear that if a microservices-based application has components that are performing integration-like work, introducing a lightweight integration runtime to the toolkit will aid productivity with a minimal learning curve.

Portability: Public, private, multicloud

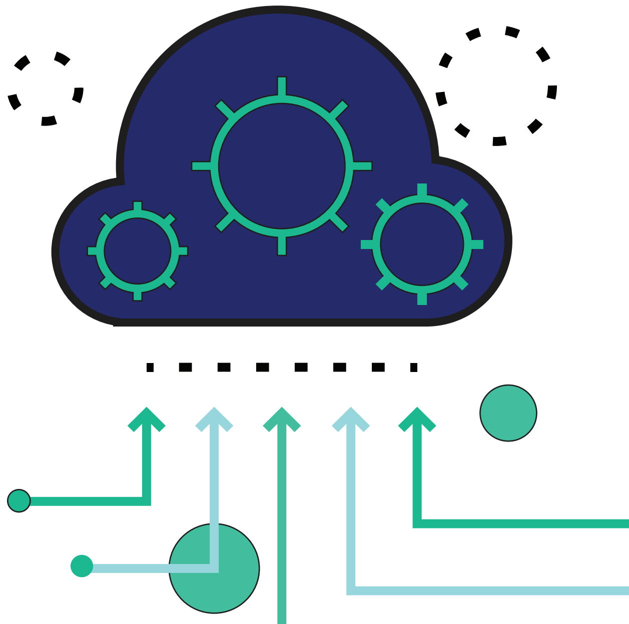
One of the major benefits of using a cloud native architecture is portability. The goal of many organizations is to be able to run containers anywhere, and to be able to move freely between a private cloud, various vendors of public cloud or indeed a combination of these.

Cloud native platforms must ensure compatibility with standards such as Open API, Docker and Kubernetes if this portability is to be a reality for consumers. Equally, runtimes must be designed to take full advantage of the standardized aspects of the platforms.

An example might be data security. Let's assume a solution has sensitive data that must remain on-premises at this point in time. However, regulations and cloud capabilities may mature such that it could move off-premises at some point in the future. If you use cloud native principles to create your applications, then you have much greater freedom to run those containers anywhere in the future.

Other examples might include, development and test in one cloud environment and production in a different one, or using a different cloud vendor for a disaster recovery facility.

Whatever the reason, we are at a point where applications can be more portable than ever before, and this also applies to the integrations that enable us to leverage their data. Those integrations need to be able to be deployed to any cloud infrastructure, and indeed enable the secure and efficient spanning of multiple cloud boundaries.



Conclusion on cloud native integration infrastructure

When we decompose what a microservices application is actually composed of, we see there is a blend of both business logic and integration. There will always be a benefit of writing integration-specific microservices in a lightweight integration runtime and taking advantage of the productivity enhancements. If we have an integration runtime available that can behave just like any other lightweight runtime, truly playing to cloud-native principles, then that's what we should be using when it comes to the many integration-centric tasks required in modern applications. It as an essential tool in the cloud-native tool box.

Lessons Learned

A real-life scenario

An organization had adopted a microservices architecture with agile methodologies. On their roadmap, this organization was on pace to build out many microservices in a very short amount of time. This notion was perfectly aligned with the attributes of microservices architecture and did not indicate any reason for concerns.

The team is new enough to plan for avoiding noisy neighbor scenarios, which would certainly lead to dependency clashes. To avoid such problems, they established the need to create a new runtime for each microservice. However, they did not choose to implement this on a cloud infrastructure. Instead, the team adopted VMs to provide this containment and required that each microservice would need to run on its own VM.

The problem

The teams immediately got stuck at a standstill, because the creation of each new service meant that they would have to create a unique VM, install a runtime on top of that VM, configure each one for that particular use case, and finally add code to that runtime. These steps would then have to be repeated and tested for each and every environment.

Development velocity came to a screeching halt as onboarding new microservices took too much time. Developers were stuck waiting for the creation of the infrastructure to run each new microservice. Inevitably, this raised the notion of leveraging runtimes that were already created. This was the exact behavior the organization had set out to avoid!

The solution

The team then realized the need for containers. A necessary component to support a microservices architecture is a cloud environment. The team quickly realized that the isolation that containers provide solved the problem of version clashes as well as isolating each individual container from the noisy neighbor scenario. The solution here was therefore straight forward - the team agreed on and adopted a cloud platform.

While this improved the situation, it didn't succeed in entirely solving the problem. The team was still treating Docker containers like VMs. The container was started with the necessary running software and dependencies, but code came and went with each new version. The concept of packaging and treating Docker images differently that VMs was lost. To improve this state, the team picked the appropriate workload and started with stateless services. From here, they could treat Docker containers like cattle, enabling a container to be disposable. They also ensuring that each new version of code resulted in a new Docker image, ensuring greater consistency between environments, and a more technology independent build chain. This provided the agility the team needed to keep up with the demands of a microservices architecture.

Section 3: Moving Forward with an Agile Integration Architecture

Now that you understand the concepts of an agile integration architecture it is important that we examine next steps. While no two journeys are the same there are some commonalities that can be explored which may help you along the path to making an agile integration architecture a reality.

- **Chapter 7: What path should you take?**
  Explores several ways agile integration architecture can be approached

- **Chapter 8: Agile integration architecture for the Integration Platform**
  Surveys the wider landscape of integration capabilities and relates agile integration architecture to other styles of integration as part of a holistic strategy.

Chapter 7: What path should you take?

So far, you have seen how the centralized ESB pattern is in some cases being replaced by one or more of the following new approaches:

- **Fine-grained integration deployment** splits up the centralized ESB pattern into more granular manageable pieces to enable a much more agile, scalable, and resilient usage of integration runtimes.
- **Decentralized integration ownership** puts the creation and maintenance of integrations into the hands of application teams, reducing the number of teams and touchpoints involved in the creation and operation of end-to-end solutions.
- **Cloud native integration infrastructure** fully extends agile integration architecture principles into the cloud native space, treating the integration runtime as a true cloud native component.

Each of these aspects is an independent architectural or organizational decision that may be a good fit for your upcoming business solutions. Furthermore, although this booklet has described a likely sequence for how these approaches might be introduced, other sequences are perfectly valid.

Each aspect of agile integration architecture is an independent architectural decision, any one of which may be a benefit to your business.

For example, decentralization could precede the move to fully fine-grained integration deployment if an organization were to enable each application team to implement their own “separate ESB pattern”. Indeed, if we were being pedantic, this would really be an application service bus or a domain service bus. This would certainly be decentralized integration—application teams would take ownership of their own integrations but it would not be fine grained integration because each application team would still have one large installation containing all the integrations for their application.

The reality is that you will probably see hybrid integration architectures that blend multiple approaches. For example, an organization might have already built a centralized ESB for integrations that are now relatively stable and would gain no immediate business benefit by refactoring. In parallel, they might start exploring fine-grained integration deployment for new integrations that are expected to change quite a bit in the near term.

Don’t worry...we haven’t returned to point-to-point

Comparing the point-to-point architectures we were trying to escape from in the early 2000s with the final fully decentralized architectures we’ve discussed, it might be tempting to conclude that we have come full circle, and are returning to point-to-point integration. The applications that require data now appear to go directly to the provider applications. Are we back where we started?

To solve this conundrum, you need to go back to what the perceived problem was with point-to-point integration in the first place: interfacing protocols were many and varied, and application platforms didn’t have the necessary technical integration capabilities out of the box. For each and every integration between two applications, you would have to write new, complex, integration-centric code for both the service consumer and the service provider.

Now compare that situation to the modern, decentralized integration pattern. The interface protocols in use have been simplified and rationalized such that many provider applications now offer RESTful APIs— or at least web services and most consumers are well equipped to make requests based on those standards.

Where applications are unable to provide an interface over those protocols, powerful integration tools are available to the application teams to enable them to rapidly develop APIs/ services using primarily simple configuration and minimal custom code.

Along with wide-ranging connectivity capabilities to both old and new data sources and platforms, these integration tools also fulfill common integration needs such as data mapping, parsing/ serialization, dynamic routing, resilience patterns, encryption/decryption, traffic management, security model switching, identity propagation, and much more—again, all primarily through simple configuration, which further reduces the need for complex custom code.

The icing on the cake is that thanks to the maturity of API management tooling, you are now able to not only provide those interfaces to consumers, but also:

- make them easily discoverable by potential consumers
- enable secure, self-administered on-boarding of new consumers
- provide analytics in order to understand usage and dependencies
- promote them to externally facing so they can be used by third parties
- potentially even monetize APIs, treating them as a product that’s provided by your enterprise rather than just a technical interface

In this more standards-based, API-led integration, there is little burden on either side when a consuming application wants to make use of APIs offered from another provider application.

Of course, API management is only part of the picture. API management provides the standardized, secure, discoverable exposure of an API, but what if the application in question doesn't provide an API today, or it does, but it's the wrong granularity, or it is overly complicated, or it has a complex security model. This is where application integration runtimes come into play. They provide the tools to perform deep connectivity, unpick complex protocols, compose multiple requests to produce an API that is appropriate for exposure through an API management layer.

It's not point-to-point because, this integration and surfacing of the API is only done once, on the provider side, for a given capability. It can then be re-used easily by multiple consumers, and its usage can be monitored and controlled in a standardized way.

Deployment options for fine-grained integration

As the organization considers shifting the architecture, there will be an inevitable question about whether to deploy the integration components on premise or on the cloud.

Many organizations are choosing both – recognizing there are scenarios that lend themselves more in one direction or the other.

Therefore, when it comes to deployment options, the integration technology must provide "choice with consistency". Consistency refers to having the same capabilities available regardless of how the platform is deployed.
In this way, the enterprise users have ultimate flexibility and avoid making trade-offs between "right architecture" versus "best productivity". Choice means that there are multiple deployment models that help satisfy organizational imperatives, which may include:

- Simplified administration and management
- Performance optimization
- Dynamic scalability/flexibility

Organizations should seek out options for a hosted service in the cloud (often referred to as an Enterprise iPaaS), an installable software image, or as a prebuilt Docker image (as we have largely been discussing). Each of these deployment options has a value that aligns to the imperatives listed above.

Depending on your specific organizational goals will lead you to choose one of these options over the other. The following three imperatives are expanded on here to help guide that decision making:

Increasingly, organizations will need to deploy integration technology in hybrid fashions and therefore need choice of deployment option and consistent functionality in all options.

Simplified Administration and Management

One of the great benefits of managed software is that it lowers the level of expertise required for anyone to be successful. This is a key concern where enterprises are looking to push the integration capabilities outside of their core IT operation. Many organizations are seeking simpler deployment, management and administration models, particularly when the workloads are not as aggressive, or where cost is a primary issue.

Where a single organization integration in multiple solutions (i.e. most businesses), that business may in fact seek to satisfy both imperatives.

In this situation, organizations may favor the managed service option. An environment can be provisioned within a multi-tenant cloud within minutes. The vendor maintains the health of the environment and currency of the software, greatly reducing the time, energy and cost of traditional server installations.

Performance optimization

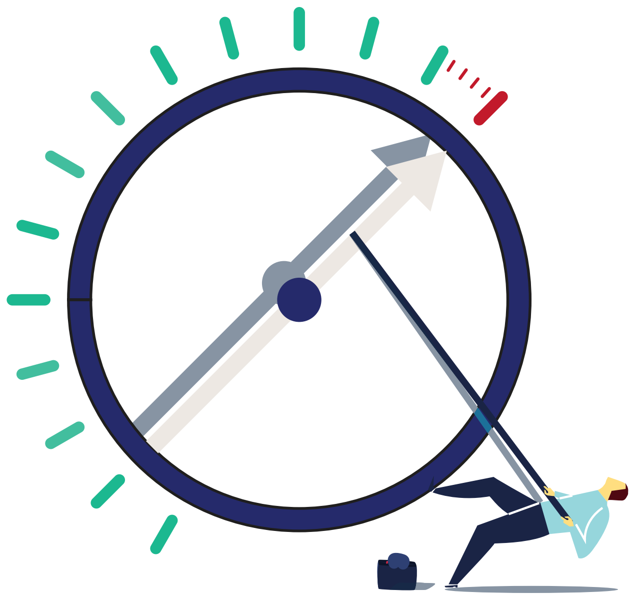
Maximizing performance is a multi-faceted requirement. Within real-time architectures, the primary consideration is typically reducing latency. In this scenario, we want the message (or service call) to execute with as little friction as possible. Collocating hardware has an advantage in reducing network hops and avoiding network congestion. Pinning key reference data in local caches provides a means of avoiding making additional external calls which themselves introduce communication time. Ensuring the service has a large enough pipe at anytime to accept any incoming requests also avoids wait times. A system that deals with such requirements effectively tends to cost more, but where the business solution is mission-critical, it may well be worth the time, effort and cost.

If performance optimization is the primary requirement, an organization will likely prefer an on-premises installation on dedicated hardware and network infrastructure. The integration platform should be installable in the hardware environments of your choice (X, P and Z hardware) – whichever best fits the solution requirements.

Dynamic scalability/flexibility

Many organizations have spikes in processing that happen at various times in the year. For the retailer, these periods occur around Thanksgiving or Valentine’s Day (or others depending on the specific merchandise). For healthcare companies, there is a tendency to see larger workloads during open enrollment periods in November and December. However, other spikes in workload cannot be so neatly planned, and when the workload represents significant business opportunity for profit, the ability to scale up processing quickly is paramount to success. In this book, we have explored the container-based and microservices-aligned architecture which is perfectly suited to helping organizations with this requirement. While other architecture choices do exist, the repeatability of the container-based model across many IT disciplines makes this increasingly attractive.

As we have discussed earlier in this book, the integration technology should be available as a container. This fine-grained deployment model removes single points of management and control so that the architecture can scale independently of other workloads in the environment. Following the principles of cloud-native applications, the technology is then a perfect fit for organizations pursuing such scalability and flexibility.



Agile integration architecture and IBM

IBM has been leading innovation in the integration space for 20 years, is a market leader for each integration capability and has been investing significantly in agile integration architecture. As such, the aspects that we've explored through the prior chapters are all areas that are supported with IBM Cloud Integration Platform.

In the following chapter, we will provide a survey of the IBM Cloud Integration Platform so that you can understand the key capabilities it offers and some of the primary use cases that customers generally apply it to. We hope that material is useful in complementing your integration strategy.

While not covered further in this book, another technology which will be interesting to organizations who recognize the merits of this approach is IBM Cloud Private. IBM Cloud Private is a robust application platform for developing and managing on-premises, containerized applications. It is an integrated environment for managing containers that includes the container orchestrator Kubernetes, a private image repository, a management console, and monitoring frameworks. IBM Cloud Private also includes a graphical user interface which provides a centralized location from where you can deploy, manage, monitor, and scale your applications. IBM Cloud Private fully supports the orchestration requirements of the approaches we have described in this book.

Chapter 8: Agile integration architecture for the Integration Platform

What is an integration platform?

Through this book, we have been focused on the application integration features as deployed in an agile integration architecture. However, many enterprise solutions can only be solved by applying several critical integration capabilities. An integration platform (or what some analysts refer to as a "hybrid integration platform") brings together these capabilities so that organizations can build business solutions in a more efficient and consistent way.

Many industry specialists agree on the value of this integration platform. Gartner notes:

The hybrid integration platform (HIP) is a framework of on-premises and cloud-based integration and governance capabilities that enables differently skilled personas (integration specialists and nonspecialists) support a wide range of integration use cases.... Application leaders responsible for integration should leverage the HIP capabilities framework to modernize their integration strategies and infrastructure, so they can address the emerging use cases for digital business[3].

One of the key things that Gartner notes is that the integration platform allows multiple people from across the organization to work in user experiences that best fits their needs. This means that business users can be productive in a simpler experience that guides them through solving straightforward problems, while IT specialists have expert levels of control to deal with the more complex enterprise scenarios. All of these, users can then work together through reuse of the assets that have been shared; while preserving governance across the whole.

Satisfying the emerging use cases of the digital transformation is as important as supporting the various user communities. The bulk of this chapter will explore these emerging use cases, but first we should further elaborate on the key capabilities that must be part of the integration platform.

The IBM Cloud Integration Platform

IBM Cloud Integration brings together the key set of integration capabilities into a coherent platform that is simple, fast and trusted. It allows you to easily build powerful integrations and APIs in minutes, provides leading performance

[3]Hype Cycle for Application Infrastructure and Integration, 2017, Elizabeth Golluscio.

and scalability, and offers unmatched end-to-end capabilities with enterprise-grade security.

Within the IBM Cloud Integration platform, we have coupled the six key integration specialties - each a best-of-breed feature in its own right. These are:

API Management

Exposes and manages business services as reusable APIs for select developer communities both internal and external to your organization. Organizations adopt an API strategy to accelerate how effectively they can share their unique data and services assets to then fuel new applications and new business opportunities.

Security Gateway

Extend Connectivity and Integration beyond the enterprise with DMZ-ready edge capabilities that protect APIs, the data they move, and the systems behind them.

Application Integration

Connects applications and data sources on-premises or in the cloud, in order to coordinate exchange business information so that data is available when and where needed.

Messaging

Ensures real-time information is available from anywhere at anytime by providing reliable message delivery without message loss, duplication or complex recovery in the event of system or network issue.

Data Integration

Accesses, cleanses and prepares data to create a consistent view of your business within a data warehouse or data lake for the purposes of analytics.

High Speed Transfer

Move huge amounts of data between on-premises and cloud or cloud-to-cloud rapidly and predictably with enhanced levels of security. Facilitates how quickly organizations can adopt cloud platforms when data is very large.

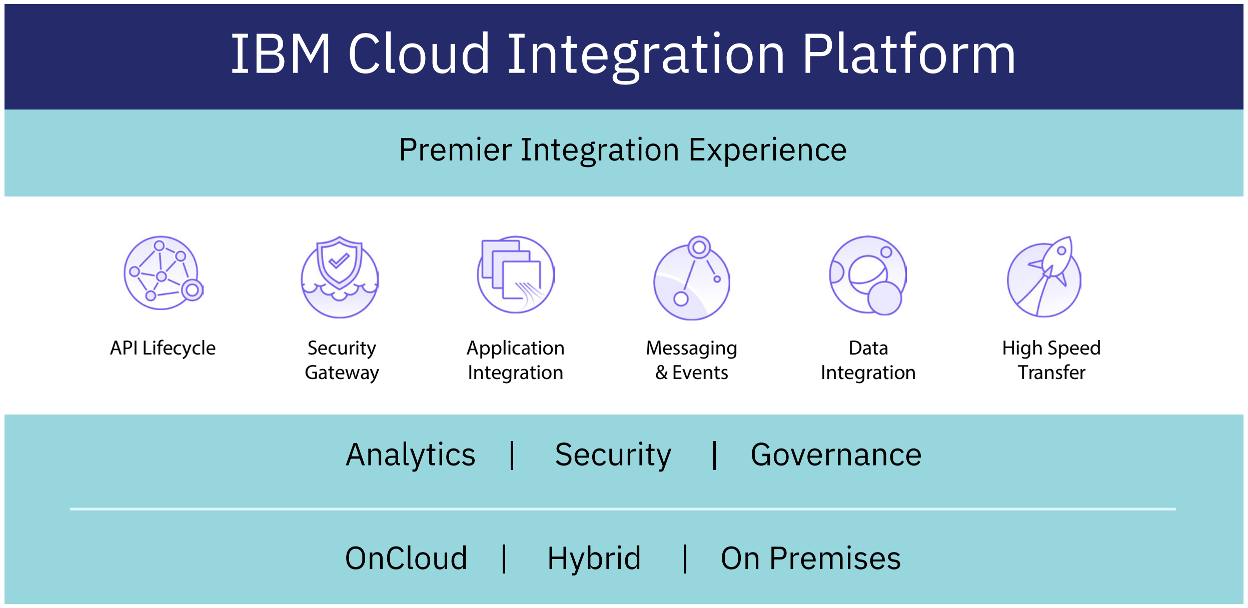

IBM Cloud Integration Platform
Premier Integration Experience
API Lifecycle
Security Gateway
Application Integration
Messaging & Events
Data Integration
High Speed Transfer
Analytics | Security | Governance
OnCloud | Hybrid | On Premises

Figure 20: The IBM Cloud Integration Platform

Emerging use cases and the integration platform

Through thousands of implementations, we have observed that customer's adoption of integration capability is normally in pursuit of very common business objectives. The four listed in this chapter are not the only relevant patterns, but are among the most pervasive across organizations of any size. After we describe each use case, we'll then also look at some of the key integration capabilities that leading IT professionals apply to be successful.

Scenario 1: Unlock business data and asets as APIs

API Management is one of the fastest growing segments in the integration space. The reason for this is based on the speed at which organizations can build new business opportunities through a robust API strategy. The ability to socialize and get applications, services, or data into the marketplace is critical for any company that wants to grow. One of the best ways to do this is by exposing services as APIs for external consumption. Organizations do this to either encourage development and expand their presence in an ecosystem, or to create new revenue opportunities by using APIs. Usage increases as organizations grow their ecosystems and as their products or services integrate with more applications and platforms. A properly designed self-service API Developer Portal allows internal developers and partners to quickly gain access to underlying apps without sacrificing security. It also socializes microservices and APIs across teams, reducing duplication of work.

API Management is one of the fastest growing segments in the integration space. The reason for this is based on the speed at which organizations can build new business opportunities through a robust API strategy.

Deciding to adopt an API-led approach is of course just the beginning of the story, you then need to actually implement the APIs. This comes in two parts:

- An outward facing **API management** capability providing a gateway to make the APIs safely and securely available to the outside world, and providing the self-administered developer portal to enable consumers to discover, explore and gain access to the APIs.
- An **application integration runtime** to enable access to data held deep in systems of record, transforming, translating and enriching the data to the point where it is fit to be exposed via the API gateway.

One of the primary drivers behind an API strategy is to encourage **innovation**, by providing external parties with the opportunity to think creatively about how to leverage your data and build it into new business models. This is very different from traditional integration where the required interfaces where often well known in advance and driven by specific projects. APIs are much more demand driven, and are constantly evolving as the ecosystem around them develops. Agile integration architecture enables us to react to this continuously iterating environment, allowing safer adjustment and introduction of individual integrations in isolation.

Also, critical to the API economy is **elastic scalability,** as it is nearly impossible to know which of your APIs will become popular. The cloud native infrastructure employed by agile integration architecture enables us to start small yet still scale on demand should a particular API start to gain traction.

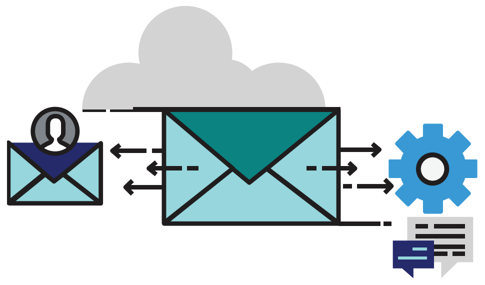
Scenario 2: Increase business agility with a modern messaging and integration infrastructure

Many enterprises have long used messaging and integration at the heart of their critical business applications. As they shift their attention to the cloud, and especially to microservices, delivery of information by messaging becomes even more important. One of the key design points of microservices architecture is that microservices should each be highly independent and decoupled, and messaging is a key way to achieve that.

However, when it comes to delivering messages across application boundaries they face some challenges. Where they would like to build new customer engagement experiences on a cloud hosted infrastructure, they are finding that tying these new systems into their existing on-premises back-ends is challenging.

This is further complicated as different parts of the organization start adopting IaaS in different cloud platforms.

While these cloud platforms may include messaging technology, IT teams are finding that the assumptions of the lower qualities of services provided by these platforms (typically “at least once delivery”) increase the burden on every new application to program to this new pattern in a consistent way. Finally, these new messaging platforms don’t naturally bridge into the existing backend systems, so integrating them across the DMZ becomes a challenge of its own. Organizations need messaging and integration platforms adept at bridging across cloud and back end systems reliably and securely.



Organizations need messaging and integration platforms adept at bridging across the cloud and back-end systems in order to provide consistent solution development experiences and speed productivity.

Modern messaging and integration middleware brings a new set of capabilities to overcome these challenges:

- Enhancement of the enterprise integration platform components to embrace cloud characteristics such as elasticity, security, scalability, and others.
- Multicloud strategy using connection and integration capabilities on external vendor cloud platforms through open standards to use best-in-class capabilities and avoid vendor/platform lock-in.

The modern messaging offering must provide robust, scalable, secure, and highly available asynchronous messaging to allow applications, systems, and services to exchange data through a queue, providing guaranteed once-and-once-only delivery of messages, enabling the business to focus on the applications rather than technical infrastructure. Ultimately, a high quality distributed messaging capability allows the application to become portable to wherever that messaging capability can be deployed.

In addition, an integration runtime then simplifies how different applications and business processes interact with the messaging layer regardless of the application type (for example, off-the-shelf, custom-built, software as a service), location (private cloud, public cloud), protocol, or message format.

Messaging is all about decoupling; isolating components from one another to reduce dependencies, and increase resilience. Fine-grained integration deployment further increases that resilience by ensuring that wherever messaging interactions require integration, they have their own dedicated containers performing that work, reducing regression testing, and improving reliability.

Agile integration architecture also simplifies migration to and between cloud platforms since the integrations relevant to a particular application can be moved independently of the others. The integrations live with the application rather than in an inflexible centralized infrastructure.

Scenario 3: Transfer and Synchronize Your Data and Digital Assets to the Cloud

One of the most critical aspects of the customer experience is responsiveness and ease. We live in a “now world” where businesses and consumers expect instant access to the information they need. The technical difficulty of providing reliable and secure access to this data does not concern them. Regardless of the communication channel, distance, or device, they expect timely and reliable information and action whenever they interact with your organization.

This need creates difficulties for organizations on several fronts. An obvious one is the delivery of any size, number, or type of digital asset to anywhere. Today, data size, transfer distance, and network conditions still greatly impact the speed and reliability that customers will get versus what they expect. This dilemma has become chronic as more industries become data-driven and operations expand globally.

Another difficulty is a bit more behind the scenes. The amount of data created for and by all of us is growing exponentially in our hyper-connected world. Today, businesses are moving to a multicloud environment to gain maximum agility, efficiency, and scale, while lowering operating risk. To support big data processing in the cloud, organizations need a solution specifically designed to move large files and data sets to and from multiple cloud infrastructures quickly and securely.

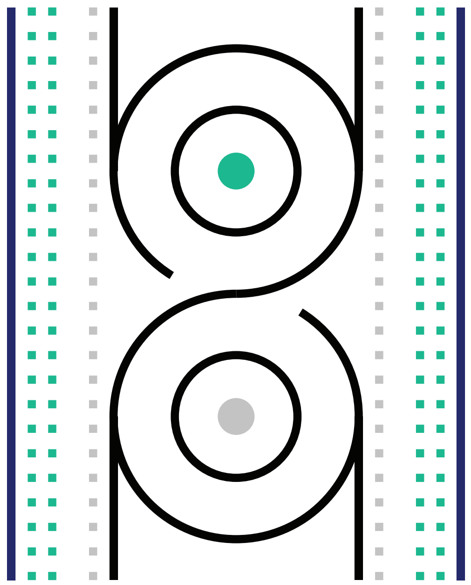
Shifting large data volumes between data centers and the cloud infrastructure can be a primary roadblock to cloud adoption unless addressed through high speed transfer technology.

IBM Cloud Integration provides a comprehensive data transfer and sync system that is hybrid and multicloud, addressing a flexible set of data transfer needs. This high-speed transfer technology makes it possible to securely transfer data up to 1000x faster than traditional tools, between any kind of storage, whether it's on premises, in the cloud, or moving from one cloud vendor to another, regardless of network latency or physical distance.

Some common situations for high speed transfer are:

- Sending and syncing urgent data of any size between your enterprises' data centers anywhere around the globe
- Sending and syncing data to any major public cloud by using our presence in all public clouds to enable cloud migration at high speed
- Participating in larger solution patterns along with other integration technologies (such as messaging and application integration) to reduce latency and provide delivery consistency

Therefore, in a multicloud architecture, particularly where part of the solution requirements is to transfer video or other large files, the ability to distribute these capabilities across the topology in a distributed manner is paramount to achieve good customer experiences. Organizations must then consider weaving high-speed transfer into API, application and messaging-led solutions. The elastically scalable infrastructure that underlies any one of these should then also account for variability in the scale out requirements of this data transfer layer.



Scenario 4: Integrating SaaS

Businesses are rapidly adopting a new class of applications in the cloud to drive business transformation - software-as-a-service (SaaS) applications. These streamline and augment activities that were previously supported by more traditional on-premises applications. SaaS is providing innovative capabilities, low costs to get started, and the ability to rapidly scale. It is for these reasons that apps like Salesforce, Netsuite, Workday, and others have become so very popular.

To maximize the impact of their SaaS purchases, organizations can't afford for these applications to become isolated. By integrating their SaaS applications with other systems and data, organizations not only realize the full range of capabilities that the SaaS application offers, but they are able to augment their SaaS purchases with other apps and services to deliver richer outcomes that drive greater productivity and operational efficiency.

Integration SaaS applications typically are provided through technology referred to as **"integration platform as a service,"** also known as **iPaaS**. Integration platform as a service provides the full gamut of integration capability with its ability to handle connectivity and integration to applications on-premises and in the cloud. The iPaaS experience is purpose-built to simplify and accelerate the activities for creating and running integrations in the cloud.

IPaaS solutions accelerate business transformation through adoption of SaaS apps via simple configuration-based approaches to integration.

As part of the IBM Cloud Integration Platform, IBM App Connect provides a range of experiences that enable organizations to rapidly configure, deploy, and manage integrating their SaaS applications with other systems across their business or enterprise.

It offers users intuitive tools and a no-code configuration-based approach, enabling them to quickly build integration "flows". These flows can address a broad set of integration requirements:

- **event-based integrations** – watch for business events across systems and then trigger downstream actions when those events occur
- **data synchronization** – ensures that data (for instance, customer data) is kept in sync across multiple systems where it is stored and maintained
- **integration services** – exposes integration logic as a RESTful end point (API) so that it can be offered as part of any business application or process
- **batch processing** – extracts a set of information from an app, database, or other data store, transforms that information into a target format, and loads it wherever required

Many organizations are looking to compliment this iPaaS capability with API Management in a few scenarios:

- Where the iPaaS is building new RESTful integration services, those APIs need to be managed, secured and governed in a manner that is consistent with other APIs developed in the enterprise.

- Some organizations have found that they need to take an active role in managing the workload they generate against their SaaS app. This may be so that they can defer API limits from those vendors or overage charges. API Management can be inserted to gate access to these SaaS apps, and prioritize certain classes of enterprise workload. Additionally, each project can be metered and usage can be tracked. This would be very useful for internal charge backs.

- Coupling the iPaaS and API Management layer can provide a more consistent abstraction layer when an organization has a variety of SaaS apps that they need to build against. Without a layer of abstraction each team would have to go through that learning curve to implement with each SaaS provider.

The IBM Cloud Integration Platform is itself written using microservices architecture. This is what enables us to bring new features to market so quickly, and manage the multi-tenant load so elastically. With the most recent release, we extended these features such that you can build integrations in the cloud that seamlessly hook into any of your enterprise systems. This provides you with a single product that has both rich enterprise connectivity along with a huge breadth of cloud application connectors, enabling true any-to-any integration on a lightweight architecture.

Conclusions

Through this final chapter, hopefully you've gotten a broader perspective of the various critical capabilities required as part of an integration platform, a sense of the requirements for those capabilities to work together, and an appreciation of how the agile integration architecture can be adopted to enable greater agility, scalability and resilience for the platform.

It is also our hope that you've gained an appreciation for how IBM has continued to innovate so that our customers can benefit from adopting modern integration technologies that assist them ultimately in satisfying their digital transformation objectives.

Kim, Nick and Tony are very happy to entertain questions, receive feedback, and advise on specifics that might not have been covered in this work. If you'd like to reach out, please find our contact information in the "About the Authors" section. Of course, we are also happy to be working for IBM where we have a great team of professionals who also stand at the ready. If you already have friends at Big Blue, we're sure they would also be happy to get your call.

Appendix One: References

New material on this topic will be published/promoted on:
http://ibm.biz/AgileIntegArchLinks

SA regularly updated collection of relevant links exists here:
http://ibm.biz/AgileIntegArchLinks

The book builds on the following source material

- Moving to agile integration architecture
  http://ibm.biz/AgileIntegArchPaper
- The fate of the ESB
  http://ibm.biz/FateOfTheESBPaper
- Microservices, SOA, and APIs: Friends or enemies?
  http://ibm.biz/MicroservicesVsSoa
- Cattle not pets: Achieving lightweight integration with IIB
  http://ibm.biz/CattlePetsIIB
- The hybrid integration reference architecture
  http://ibm.biz/HybridIntRefArch