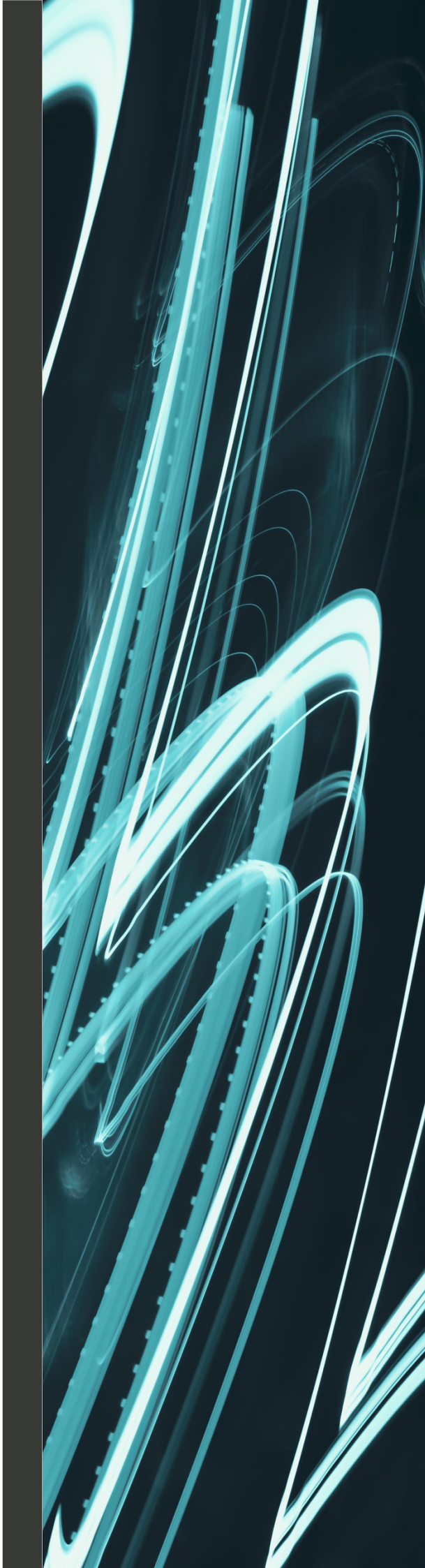
responsiv

simple · effective · distinctive

WHITEPAPER

Digital-SOA: Making the Best of an ESB



Preface

Integration of enterprise systems has been a challenge for many years. Initially systems operated as islands of automation, connected only as an exception. Integrated systems are now a primary requirement and Service Oriented Architecture (SOA) is a preferred style. SOA promises re-use, location independence, and the benefits of mediated integration. Determination of scope and behaviour that should be provided by an "SOA interface" has regularly undermined the expected value from SOA.

Digital-SOA takes the SOA model to a new place, one designed to accommodate bi-model IT, Omni-channel, declarative interfaces, mobile technology, and customer-centricity at its very core.

This paper takes a look at the challenges of extending the basic SOA architecture to accommodate the new demands of Digital platforms. Specifically, it considers what additional capabilities are essential to properly support a digital enterprise.

Intended Audience

This paper is intended to support executive and leadership teams in consumer-focused companies to understand what Digital-SOA and Digital economy really mean, and how they must be addressed.

Disclaimer

Observations and recommendations documented in this white paper are based on our opinions, experience, and research. They are as objective and representative as we can reasonably be, however Responsiv makes no representation as to accuracy or fitness for purpose. Once you have chosen a course of action, it should be thoroughly evaluated to ensure its fitness for purpose.

About the Author

Richard Whyte is an accomplished IT architect with a proven ability to innovate and focus on customer requirements to deliver simple, effective solutions. He has demonstrated thought leadership based on a breadth of technical and project experience spanning Investment Banking, Retail, Manufacturing, and Aerospace, delivering sustainable technology to some of the world's largest companies.

Richard has a degree in statistics and computing and a master's in business administration. He is a Fellow of the British Computer Society (FBCS), Chartered Engineer (CEng), Chartered IT Professional (CITP), and Fellow of the Institute of Engineering and Technology (FIET).

About Responsiv

Responsiv Solutions is a UK based company that specialises in delivering business integration across the enterprise, including API management, Business Process Automation, and Digital-SOA platforms. We work across many industries, including Retail, Financial Services, and Government.

Responsiv can provide fully commissioned solutions that include all the software and professional services needed to deliver an integration platform to support your business plan and grow with your business.

Table of Contents

Introduction

Using an SOA style places a significant reliance on re-use and shared infrastructure, which changes the way an enterprise considers its processes, how it finances and controls technology and how it is organised to operate and support its infrastructure. Key success criteria include:

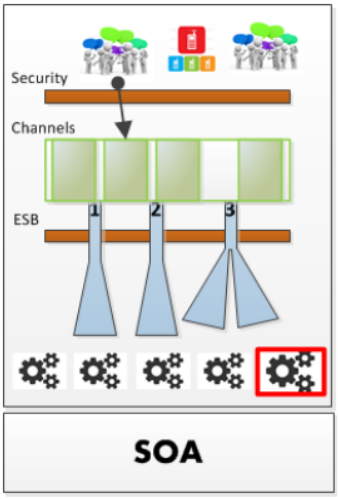
- Appropriate governance and decision model
- Appropriate financial and chargeback model
- Appropriate and empowered operational support organisation
- The objective for this report is to inform decisions about how to design and structure a Service Oriented Architecture (SOA) and the role and expected capabilities of key SOA components.

Structure

The report is structured to introduce the SOA style with its key principles and objectives, strengths and weaknesses. Other integration styles are described in brief for comparison. A logical model is used to locate each of the key components and justification for being in the platform. The following sections detail each component in terms of its expected capabilities with justification, including a set of assessment criteria to facilitate product selection and development priorities.

SOA Principles and Concepts

> "Moving to SOA is no small feat. It can and should be done incrementally, but requires a shift in how we consider computer resources while maximizing existing technology investments."[1]



Service-Oriented Architecture (SOA)[2] is a style of architecture that seeks to focus provision of functions and data, as well as all deployed artefacts, on elements known as "services". Each service is represented by an implementation and an interface, the SOA reference model insists that each implementation is independent of all others, and that its interface is unique and independent of all others. Typically, an SOA is based on a layered architecture approach with consumers calling through a mediation (ESB) to providers.

Key principles are:

Independent implementation: Each service is independent of all others. This means that functional changes, capacity management, retirement, and location changes are all possible with minimal testing and low risk of unintended side effects. No artefact (message, message structure, utility class) can exist in the SOA environment that is not either a service, or owned by a service. Disaster recovery, impact analysis, testing, and agility are all improved over architectures without the constraint, and benefits are realised to the business including faster implementation of changes, quicker recovery from failure and more accurate and predictable testing.

Separation of concerns: Each service and its implementation are deployed separately from one another. This means that an interface can remain in a known place when its implementation moves, horizontally scales, or is replaced with a different package. It also means that implementations that are deployed in two or more applications can be aggregated without involving the service consumer.

Isolation: The service interface and ESB provide high degrees of isolation and direct interface styles toward loose cohesion and coupling. Isolation is not limited to the protocol, it should extend through all communication layers to include data representations and interface data structures. The service interface is an important point of control and isolation only if it is established as the single point of contact between one service and the next; this does not prevent multi-protocol, multi-lingual interfaces built onto a standard (internal) point of control.

- "Backward compatibility"[3] the Service interface is an important aspect of isolation and context invariance as well as further isolate other services from changes to the implementation. This constraint requires strong interface design work at the start but is not as difficult as it appears.
- Consistent and predictable interfaces reduce errors and improve the ability to change implementation without leaking details through the interface.
- Separation of sections of the interface payload according to owning service provides for adherence to the SOA principles and supports change and governance.

[1] Anon, A Technical White Paper June 2004; Assessing Your SOA Readiness, SUN Microsystems Inc. (2004)
[2] The term "Enterprise" is used as a convenient generic term for any reasonably self-contained commercial, technical or other organization. Enterprise architectures and solution frameworks must be capable of addition and removal of previously key components without major engineering works, and to sustain those abilities for longer than the lifetime of the people and products involved with the initial construction.
[3] Backward compatibility implies that versions of interfaces do not offer different capabilities, which otherwise leads to proliferation of interfaces with subtle differences that are expensive to test, and undermine principles of backward compatibility.

Specialisation: Each service must specialise in one task or activity. This means that the functions available from a service are clearly identifiable, and that the scope of a service can be constrained to as small as necessary to encapsulate the implementation. Services should specialise in delivering a single, specific capability to the platform. This means that the scope can be well defined and understood, and that the internal implementation does not become so complex and extended that the benefits are lost.

"Context invariance" assures consumers of the service that they will get the same result from the same call regardless of their context. In this case context includes the pattern of previous calls, the origin (mobile, web, internal) of their call, and the time of day. This does not impose repeatability on the call because while the interface behaviour does not change the underlying data might.

In summary a service is constructed from an interface and implementation and is subject to independent implementation, delivers a single capability, its interface is backward compatible and context invariant.

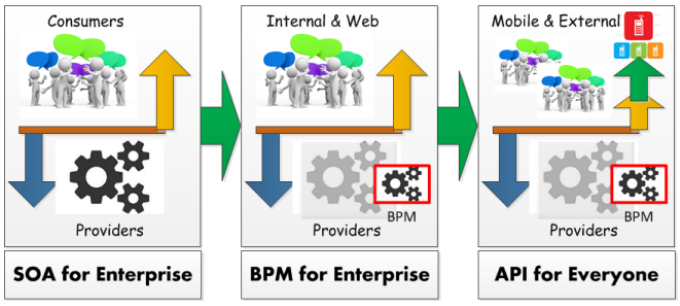


Figure 1; Context invariance

The SOA model has evolved to accommodate new challenges and deliver control and accessibility as new technology has become available. In the early days the SOA model was used to separate consumers from provides in a pure internal sense, and to deliver the benefits of applying the principles of SOA to the enterprise.

The addition of BPM mirrored the car-industry example to add efficiencies and new capabilities and business capacity to the basic SOA platform.

The modern SOA deployment includes API management to promote only those interfaces that are applicable to mobile and remote systems in a manner that makes sense to those consumers. The API gateway provides monitoring, throttling, and security that is not always needed internally and does not perform the "heavy lifting" expected from the ESB.

ESB Services Focused on Business Entities

> Cars were assembled at a single workstation (application); runners brought parts in order of manufacture, and the fitter fitted the parts. The assembly-line (ESB) quadrupled output by moving work-items between workstations already stocked with one type of part. Automation (BPM) improved the line by a further eight-fold using conveyor belts.

SOA Service interfaces are designed to be consumed by business processes and support business strategy, they are designed for consumers (outside-in) and business agility rather than "inside-out" to provide access to the limited functions of an application. Each service should be named for its specialised business recognisable activity to make sense in a business (nontechnical) discussion about a process.

The principles already outlined have several implications

- Implication: The possibility of an interface aggregating data/function from several different sources, and to provide the ability for implementations to scale and move without notice means that the interface MUST be deployed and controlled separately from the implementation.

- Implication: To successfully isolate the implementation a service boundary must completely encapsulate and isolate the implementation from all other services. This means that a service "update_customer" cannot exist because the data could never be read; an operation customer_update conforms to the principles.

ESB Service Operations Focus on Activities

The service name indicates the capability and boundary while the operations suggest the action or feature of the capability that is required. Two basic styles for naming of operations are used:

- Get-Put-Do styles use a selected verb to indicate modification of state, reading of state, and performance of a function, for example calculating a rate. The style is a derivation of the HTTP and object-oriented programming verbs and very common. Unfortunately, the result can be a proliferation of operations on a service that are sometimes difficult to predict.
- CRUDS style use the words (verbs) Create, Read, Update, Delete, and Subscribe to describe what is wanted and then allow the data and internal functionality to inform the actual activity that might be executed to return the result. The style is taken from SQL and relational databases and used in REST and other common styles. The operation does not provide as much of a clue to the action but the flexibility and security models can be a great deal simpler.

Interface Granularity

To assist understanding this document refers to three dimensions of service granularity; width to indicate the number of fields in the interface, breadth to indicate the range of operations, and behaviour to indicate how the service responds to exceptions and introspection. Characteristics that impact the usability and cost of ownership for a service include:

Breadth: Schema Size

The size of schema (fixed) will determine the amount of work needed to populate, validate, and move the data around. Larger schemas will likely require visits to multiple providers and may cause buffering problems when repeating groups are allowed.

Smaller schemas for services that are not dedicated to a particular consumer tend to require more calls and increased parsing and validation load as well as imposing cross referencing and unique key management activity on the client.

Behaviour: Number or Versions

The number of differently behaving versions of an interface that are active at any time will drive cost and complexity as well as confusion and increased scope and impact of change. Interfaces may be version controlled for audit and management, for example to understand the release characteristics and added functions of later versions. Version control should not be used to distinguish interfaces that have functionally different behaviour.[4]

- Avoid variants that use version numbers to distinguish different behaviour
- Use version numbers to track and control upgrades, extensions to function, and error resolution

Overlapping Interfaces

Implementations that have multiple interfaces that include the same fields as one another, or that provide different ways to access the same data (see below for version and immutable techniques) cause increased programmer confusion, the cost of testing and impact of change to increase, and from experience tend to leak details of the provider implementation. Overlapping interfaces indicate that the interface or implementation is not independent.

[4] Functionally different in this context indicates that two versions of an interface provide materially different responses to the same request and require different handling from the consumer.

Security

Most organisations still secure the interface based on service and operation name rather than the data fields themselves. The result is that it is difficult to secure confidentiality of sensitive fields that are embedded in larger schemas that cannot be redacted for some reason. It also means that the security model is securing the same data several times because it is represented in several interfaces, rather than once as a field.

Architecture Style Comparison

> SOA can be considered a specialised subset of a component architecture. It should be compared to other architecture styles rather than limited to the integration capabilities it delivers.

Component architectures consider each component to deliver a specialised capability, and construct higher order functions from wiring them together. Typically, a component architecture does not force a hierarchy or standards onto the components, which can be wired in any order and have multiple interfaces. Component linkages can range from tight to loose, and components may have recursive relationships.

Layered architectures impose a structure on the deployment of components such that a component in a layer may not call a component in a higher layer and may only call components in its own layer and one layer below. The result is that dependencies flow down the layers and a given layer can be easily replaced. Failures and changes are simple to test.

Event architectures generally consist of components that do nothing until triggered by the receipt of an event. Each event must be designed to be independent of other events and have no explicit affinities with particular times or technologies. Events may be processed through a component or layered model in a synchronous or asynchronous manner.

Batch architectures operate to process collections of data-records at the same time. Batch integration moves data between independent systems in collections to synchronise or allow that data to be processed. ETL tooling and SQL bulk loaders are typical of batch architectures. The batch style is very useful when horizontal scaling is not possible, or when data must be processed in a specific order. In every other case the event style should be closely considered.

Mediated Integration architectures require that communications between components of the architecture are brokered through a mediator. In the case of SOA we term the mediator an "Enterprise Service Bus" but any hub and spoke or adapter pattern can be considered to provide mediation between components.

- Distributed mediation (adapters without central bus) can be implemented using several Java frameworks and is a popular choice when shared infrastructure is not possible as well as for highly resilient military deployments where single "king of the mountain" components are likely to cause constraints and points of failure and cost.
- Centralised mediation (adapters and/or central bus) are used when the cost and complexity of fully distributed is not tolerable. This model is simpler to maintain and operate but can also have a distributed deployment of the central components to mitigate single points of weakness.
- Unmediated Integration architectures (point-to-point and pure publish-subscribe) require that all components are able to communicate directly (network topology and adapter languages/structures). They tend to be quite tightly coupled even when pub-sub is used.

Service oriented architectures are characterised by a focus on services, the way they interact and are consumed. This far-reaching interest goes beyond integration to address the impact of change, business continuity, ownership, and opportunity for isolation and separation of concerns. SOA is a distributed, event driven model that provides organizational and technical benefit by simplifying services through specialization and encapsulation of complexity.

The SOA model adds constraints to the standard component model by restricting the way services interact, organisation into layers, and standardisation of interface. To properly conform to the principles of SOA the architecture requires a mediation service that behaves as a proxy for all other services; in SOA we refer to this as the ESB.

Consumers know and call the ESB but have no knowledge of the underlying service implementations. At one level this is sufficient to claim an SOA deployment, however if the implementations are not also independent then many of the benefits of SOA are lost.

- Atomic services are those that deliver their function/data independently of all other services
- Assembly services deliver their function/data by orchestrating other services

Conclusion

We have considered SOA in the context of the enterprise and other architectural styles and (hopefully) concluded that the benefits outweigh the costs and frustrations of building the platform. We should also have seen that as an enterprise we are capable of delivering the governance and changes to culture needed to make the endeavour a success.

The next section details a typical logical architecture outline and the key components of a modern SOA architecture.

Key Components of SOA

We have considered SOA in the context of the enterprise and other architectural styles. In this section we propose a strawman vanilla deployment as a reference to place the key components of a modern SOA architecture.

The structure and parts of an enterprise platform must support the purpose and business case used to justify the cost of its build and operations. The structure for a "standard" deployment for SOA is a layered framework that allows requirements and dependencies to flow in one direction down the stack. Each layer is isolated from the next using gateways or the service bus, and the type of data and function in each layer is categorised to optimise the scaling and availability models.

- Products and home-built components are selected to support the structure and deliver required capabilities
- A capability may require more than one product to be bound into a service
- A single product may be deployed to several services to provide different functions
- Products should be limited to purpose and scope for which they were procured, not for their total capabilities

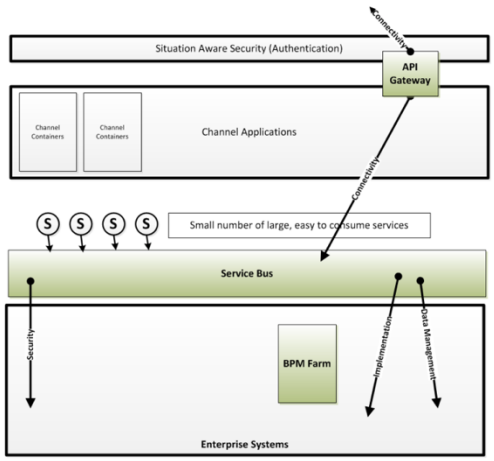
Components (services) can be categorised in many different ways. One dimension is whether a service provides structural integrity to the platform with sub-categorisation of control (security, policy enforcement point, core security), Platform (DevOps, Deployment, Monitoring)

Structural	*Description*	*Classification*
Service Bus	Separates consumers and providers, adds audit and policy enforcement, and structures integration across the platform	Structural, Integration, Separation
API Gateway	Promotes service interfaces to external consumers (inc. mobile). Supports market differentiation, throttling and protection of internal services, protocol translation, and developer support.	Structural, Security
API Manager		Structural, Management/Control
Security Devices		Management/Control
Process/Case Management		Platform
Mortgage System		Content
Current Account Ledger		Content

SOA Reference Outline

In the standard reference model the Service Bus is used to isolate enterprise systems (providers) from channel and external applications (consumers).

The API gateway is used to control external access to the platform, and BPM is used to automate case-management and process orchestration.



Service Bus

The SB provides a coherent backbone to which services are attached to implement highly pervasive and complex business processes. It is a service in its own right; one that wraps other services in added value functions and provides structural integrity to the platform.

While SOA brings many benefits, including high levels of reuse and return for investment (ROI) it introduces governance and management challenges not encountered with islands of automation, "dumb" integration or batch processing approaches. Specific challenges include interaction policy, performance, service level agreements, cross line of business (LOB) prioritization for change, financing, and cost of service.

The table below suggests characteristics that are desirable for a service bus to offer and a description. Recommendations are given for when one or more of the Responsiv products can fulfil the requirement.

[1] Responsiv Unity Integration Module

This is a self-hosted module that provides integration services. It is generally deployed in a paired cluster for resilience and managed by the Responsiv Unity Console Module. Responsiv offer advanced support services to develop and support service busses developed on this software.

[2] Responsiv Cloud Connectivity Platform

This is a cloud platform (PaaS) that includes separate development, testing, and production service bus environments. This is 100% compatible with the Responsiv Unity Integration Module and the two products can be clustered to share information.

[3] Responsiv Cloud Automation Platform

This is a cloud platform (PaaS) that includes separate development, testing, and production service bus AND process./case management environments. This is 100% compatible with the Responsiv Unity Integration Module and Responsiv Unity Process Module and can be clustered to share information.

Characteristic	*Description*	*Recommendation*
Isolation	The SB must be capable of isolating users of the bus from different protocols, syntax, and semantics used by systems of record and systems of interaction or those of your partners.	[1] or [2] or [3]
Protocol	Ability to translate between protocols allows a consumer and provider to use their chosen technology. Includes publications that subscribers can access using their preferred protocol: JMS, MQ, HTTP(S), (S)FTP(S)	[1] or [2] or [3]
Syntax	Ability to transform between XML, Fixed-Format Binary, EDI, and others	[1] or [2] or [3]
Semantic	Ability to reference external or internal sources to transform between different representations of the same meaning, for example the number 1 in a context is equal to "STG" or "GBP" in another.	[1] or [2] or [3]
Interaction Style	Should include Asynchronous and Synchronous communications	[1] or [2] or [3]
XA Compliant transactions	The XA protocol is used by databases and other transactional products to protect information from inconsistent updates, component failures, and properly isolated reading. The ESB will be responsible for updating multiple end-point systems that must be synchronised, as well as needing to assure that the ESB managed instruction is applied only once. A example of this might be applying a message request to a database once and only once. Without XA this can be a significant problem with the need to either update at least or at most once with reconciliation later. This causes race conditions and other challenges.	[1] or [2] or [3]
Independence		[1] or [2] or [3]
Time	Store and forward technology (JMS, MQ) allows the SB to push data toward a system when the system is unavailable or busy without being held up by the exchange. Alternatively, consumers can make requests of the SB at times when it is already busy.	[1] or [2] or [3]

	Store and forward also gives the SB an opportunity to prioritise work and not reject connections when a synchronous approach is overwhelmed.	
Location	Service bus performs routing and navigation to hide provider and consumer location details. A consumer needs to know where to find the service bus and does not need network connectivity directly to the provider system(s). This simplifies consumer (user experience) software development, improves reuse, and reduces testing and inconsistent behaviour.	[1] or [2] or [3]
Technology	The service bus needs to be agnostic of the technology connecting to send or receive data. This means that it should support as many protocols as reasonable even when a project may only need a few. Experience dictates that more will come along later.	[1] or [2] or [3]
Performance	Performance is measured as the throughput per core and the latency at a given throughput.	[1] or [2] or [3]
Performance	Using synchronous connections causes the fastest system to slow to the rate tolerable by the slowest system. It means that security protocols are weakened to the weakest member of the communication. The service bus mediates these problems so that all connected systems can operate at their optimum performance.	[1] or [2] or [3]
Caching	Distributed caching, cache management, and read/write support allows the SB latitude to reduce the impact of high demand on back-end systems of record. It allows intelligent storage of information for consistent performance throughout times of peak demand.	[1] or [2] or [3]
Horizontal Scale	Ability to scale out by adding new instances of the SB software means that performance can be sustained as the demand grows, and the reliability of the SB can be properly managed by locating SB instances in areas to avoid shared risks.	[1] or [2] or [3]
Vertical Scale	What is the performance curve when adding cores to a single instance. Most software scales to a point (about 6-8 cores) before the added value of new cores is reduced. Software needs to scale in a predictable manner, and to be of sufficient quality that it does not bloat its resource use over time.	[1] or [2] or [3]
XML Parsing	Just in time and intelligent XML parsing has become a significant performance gain for environments that use XML standards that have large, sometimes sparse, XML schemas. This means that the XML is parsed just in time; if the field referenced in the first field, then that is as far as the parse goes. If the parse is in a section later in the schema, then intervening sections are ignored until they are referenced. Obvious infinite XML and XML hacks should be detected.	[1] or [2] or [3]
Ops Tooling		[1] or [2] or [3]
Remote Deployment	Central control and remote deployment is a good style for service busses that must be highly available and perhaps deployed across multiple locations, including clouds.	[1] or [2] or [3]
Graphical Console	Having a console that is simple to use and provided context and support to information can be a significant cost reduction for businesses that require 7x24 or don't have large teams dedicated to the SB technology.	[1] or [2] or [3]
Deployment Support	The SB must validate deployments and where reasonable it should allow deployments without loss of service. Round robin and live updates are two techniques but there are more.	[1] or [2] or [3]
Debug Support	Graphical distributed debugging	[1] or [2] or [3]
Tracing and Audit	Ability to trace business and technical transactions as they pass through the entire integration landscape.	[1] or [2] or [3]
Accounting	Ability to assign resource use to specific areas of the business.	[1] or [2] or [3]
Upgrades/Patching	Ability to upgrade and patch the basic SB without downtime	[1] or [2] or [3]
Integration and Monitoring	Instrumentation.	[1] or [2] or [3]
Dev Tooling		[1] or [2] or [3]
Graphical	The tooling should offer graphical development as well as coding to allow developer to learn the technology and get a fast start without inhibiting experts. SB with only graphical tooling should be avoided as being too limited.	[1] or [2] or [3]
DFDL Support	Data Flow Definition Language (DFDL) is a declarative way to describe data and read or write data structures, including files, records, and messages. It is "super powerful" and should be supported in any modern SB.	[1] or [2] or [3]
Speed to Develop	The time to develop and test integrations and new services will determine whether the business has to wait for benefits and the cost of doing marginal improvements. Faster	[1] or [2] or [3]

	development leads to a more agile business that takes best advantage of the technology investment.	
Skills Availability		[1] or [2] or [3]
Cross Training	SB products that use skills that are available AND easy to teach are more likely to be cost effective in the longer term. For example SQL and variations of the language are more likely to be available and understood than reams of Java code.	[1] or [2] or [3]
Education Availability	Training courses and mentoring is a must.	[1] or [2] or [3]
Strategic Stability	What is the outlook for the vendor and what are the risks.	[1] or [2] or [3]
Product Reliability	E.g., An old product that is actively developed and strategic part of the portfolio	[1] or [2] or [3]
Cost of Ownership	Predictable cost of ownership. COO is determined by the cost of development, ability to leverage the investment to reduce operating costs, and the reliability of the software. Any capital cost of purchase is reduced year on year by the benefits and is likely to be a small part of the cost (less than 50%) over 5 years. Subscription software should be evaluated in the same way.	[1] or [2] or [3]
Security Integration	The service bus should integrate with your security environment to support standards like OAuth and single sign on.	[1] or [2] or [3]
Business Process and Case Management		[3] Only

Business Process and Case Management

Responsiv considers that the requirement for business process management and case management is significantly different from hosting integration services in a service bus. These capabilities require extensive database services to manage the state of the process, and each process or case can proceed for minutes, months, or years. Integration logic is expected to complete is seconds or less.

The relationship between a service bus and process/case automation is not always clear. In our view the process/case automation platform should, in a layered architecture, be located alongside the systems of record, or alongside the service bus itself.

Regardless of positioning the process/case platform should not integrate directly with other systems. It should always use APIs on the service bus.

Process/Case below the Service Bus

This means that data being managed is protected by the service bus, and that the automation can use services located on the service bus using a call-back pattern. The processes are accessed using an API located on the service bus (making it consistent with other systems), and the security and audit etc. provided by the bus can be leveraged.

Processes that require ESB services access them using the bus address that is provided by the ESB when it triggers the process.

One difficulty is when human tasks are implemented in the process engine and need to bypass the bus to be accessible from the user channels.

Process/Case alongside the Service Bus

One solution to the human task problem is to move the process engine to be alongside the service bus. This is a reasonable move but can undermine the layering of the architecture if channels are allowed to directly access process APIs without going through the service bus. This is perhaps a governance problem rather than a technical one.

Exception Management

Exception management for business problems encountered on the service bus during integration activity, or in the process/case engine should be directed to an exception process. The exception process can then route the exception to a specific process for that situation or provide a general exception management capability.

This is different from technical problems handled by the IT operations team.

API Gateway

Responsiv considers that the requirement for internal APIs is significantly different from external APIs in the maintenance challenges, data formats and scope, and performance/priority needs of the APIs.

The principle of "separation of concerns" suggests that APIs presented externally should ideally be hosted in a different place with slightly different arrangements for security and management from internal ESB installations.

Externally facing APIs should be integrated into the SB without exception, and the only actions above hosting that the API gateway should perform is to transform intrinsically between the external and internal interfaces.

All enrichment, validation (beyond security), and aggregation of data should be performed in the service bus.

This leaves the gateway with a defined set of responsibilities:

De- Coupling/Isolation

Present a standardised API to external users that only contains what is needed and that hides all implementation details (aka Federated/Composite Interfaces), for example does not expose SAP or other internal identifiers and that maps to a single Service Bus API. The result is that the SB API and interface logic in the SB can properly accommodate changes to the systems landscape without change to the external APIs.

Cost of change for external APIs and the number of effected parties is massive and undefined. The idea that old versions of APIs can be maintained is unsustainable in the long term because it increases testing costs and complexity as well as extending the cyber attack surface.

API Registry Policy Management

Internal APIs generally have one class of user, and that user is trusted to follow the rules. External APIs cannot trust the consumer and have no idea of their motives for using the API. The API gateway must allow security and load shaping policies to be assigned to specific groups of users to allow them to be properly prioritised and managed.

API Flexibility and Scaling

API gateways that can integrate directly to a message queueing system have the advantage that their transaction length is shorter than synchronous protocols, and the message can be protected from failures and processing problems more quickly. It should be possible to create APIs that support real time (near real time) interactions as well as passing larger streaming objects like files. Support near real time request response to maximise end user system responsiveness. Support message batching/de-batching

Audit and Analytics

Audit data and analytics can be useful to a SB and API gateway; however the API gateway may make more use of the monetisation aspect of this data gathering, as well as the possibilities of intrusion or non-standard use of the APIs.

References

- Chloe Green (March 2015)," Revealed: The true scale of legacy IT", Information Age.
  - http://www.information-age.com/it-management/strategy-and-innovation/123459130/revealed-true-scale-legacy-it
- Accenture, "Digital Disruption in Banking: Consumer Expectations vs CMO Realities",
  - http://www.accenture.com/microsites/everydaybank/Pages/consumer-expectations-cmo-realities-infographic.aspx
- Disruptive Technology and changing customer behaviours: A perspective on banking, RBS, May 2014
  - http://www.finextra.com/resources/feature.aspx?featureid=1940
- Barbarians at the gate : the fall of RJR Nabisco / Bryan Burrough and John Helyar. New York: HarperPerennial, 1991. 338.83664 BURR
- Breakpoints: how managers exploit radical business change / Paul Strabel. Boston, Mass.: Harvard Business School Press, c1992. 658.406 STRE
- Corporate restructuring: managing the change process from within / Gordon Donaldson. Boston, Mass.: Harvard Business School Press, c1994. 658.16 DONA
- Banking on a digital future, RBS,
  - http://www.rbs.co.uk/content/corporate/downloads/5193%20RBS%20Perspectives%20Digital%20Future_v4.pdf
- Bank Reputations
  - https://d25d2506sfb94s.cloudfront.net/cumulus_uploads/document/uazu2beppv/Bank_Reputation_Tracker.pdf
- Will Mondo be the first Digital Bank
  - http://techcrunch.com/2015/05/24/mondo/?ncid=rss&utm_source=feedburner&utm_medium=feed&utm_campaign=Fe ed%3A+Techcrunch+%28TechCrunch%29

Appendix: Applicability of SOA to Business

A perfect solution can become irrelevant or detrimental if fails to operate and adapt in harmony with enterprise and its point-in-time imperatives. BDO Stoy-Hayward5 consider that the imperatives of an enterprise evolve through six stages. From the SOA perspective this translates to a time when SOA is simply not appropriate through to it becoming an essential enabler.

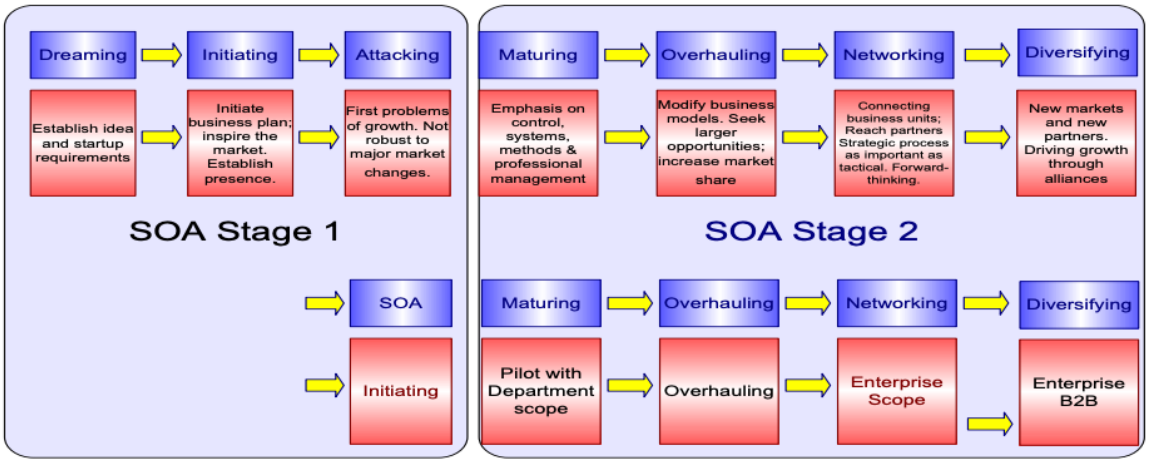


Figure 2; After the 'BDO Stoy Hayward Model of Business Growth'

Stage 1 – Attacking: Full implementation is likely to subvert business imperatives, be expensive and ineffective. Attempts to "future-proof" at this stage are subverted by rate of change of technology solutions and business needs, making them a costly mistake.

Stage 2 - Maturing: enterprises establish systems and methodologies to give them control and direct growth toward their strategic goals. This is the earliest that concerted SOA departmental effort is likely to be appropriate and may be a critical learning exercise before attempting to connect the corporate enterprise.

Stage 3 - Overhauling: SOA is directly applicable to these imperatives and most relevant and vital to the corporation. The mechanism of a process (Means) and volume of work will change but it is unlikely at this stage that business objectives (Ends) will change before the diversification stage.

Appendix: Comparison of Interface Versioning Techniques

Techniques to manage changes to interfaces are listed below, however the structure of the interface and ownership of different data in the exchange is also highly influential.

Technique	*Description*	*Recommendation*
Version the Interface	Callers must specify the version when calling. Different versions deliver different responses and cannot be retired. - Proliferation of versions - Increased testing effort - Increased confusion from programmers - Secure by Operation; difficult to redact data	A common pattern used by many companies. Considered unsustainable because old versions cannot be retired and the test/confusion simply increases over time. Use for: EAI, P2P, API6 (Recommended if backward compatibility is possible)
Immutable Interfaces	Interfaces do not change once they are deployed. If a new requirement is found, then another service with a different name is created. The approach is similar to versioning interfaces with different behaviours; both try to reduce change on the consumer side but the cost of that is to radically increase confusion for new development and cost in the provider side - Proliferation of similar services - Increased testing effort - Increased confusion from programmers - Secure by Operation; difficult to redact data	A common pattern used by many companies. Quite common in companies adopting a "purist" micro-service approach. Considered unsustainable because similar interfaces cannot be retired and the test/confusion simply increases over time. Use for: EAI, P2P, API (not practical)
Version the Request	Callers make a "versioned" request to indicate the formatting and standard used to describe their needs as well as to indicate the type of response that they will understand. - Avoids proliferation to a good extent - Clarifies testing and interface selection - Secure by Operation/version; difficult to redact data Assumes linear extension of function based on the payload version – everyone in version 3 gets all of version1 and 2 function as well as v3.	Used in many standards-based interfaces. Works very well when the behaviour of an interface always extends and builds on the previous release. More sustainable than immutable or versioned interfaces because it maintains one interface and allows it to evolve. Use for: EAI, P2P, API, SOA, wAPI (recommended to provide compatibility)
Version Request and Assertions	A variant of the request versions where the format and interpretation is based on the version but behaviour is controlled explicitly using assertions. - Provides excellent backward compatibility - Allows testing to explicitly target specific changes - Delivers good clarity to programmers - Secure by Operation/version/assertion - Data redaction possible	Technique very common in operating system interfaces and macro languages where a bitmask is used to describe assertions. Very powerful with good sustainability & backward compatibility Use for: EAI, P2P, API, SOA, wAPI (recommended)
Tagged Request with Assertions	Extension of the versioned request but tagged to allow better granularity of the format and expectations. This approach supports multiple dialects and standards through the same interface. Given that these are often very mechanical transformations it is not worth having a different interface for XML and JSON, or ISO8583 and ISO20022 - Provides excellent backward compatibility - Allows testing to explicitly target specific changes - Delivers good clarity to programmers - Secure by Operation/version/assertion - Data redaction possible except with some standards	This extension of the version+assertion approach supports a multi-standard interface as well as multiple interface languages (for example canonical models) Very good backward compatibility. Use for: EAI, P2P, API, SOA, wAPI (recommended)
Fixed Schema	A fixed schema is defined using XML, fixed-offset, or similar definition. It may be type controlled or simply define the layout of the request/response. Fixed schemas are assumed by many to be efficient and to facilitate validation and control. - Sometimes simpler to use with tooling - May simplify the way programmers think - Secure by Operation; difficult to redact data	Fixed format interfaces tend to be inflexible and have led (in our opinion) to the question of micro versus macro interfaces. They have their uses and at one time were the only option. They can bring efficiency but modern parsers and self-describing languages bring other perhaps better options. Use for: EAI, P2P, API, wAPI (not recommended for SOA)

Self-Describing Schema (Declarative)	Self-describing schemas include XML and EDI based schemas but are specifically able to adjust their response characteristics to the nature of the request (instead of simply filling a fixed schema). - Opportunities for provider-side optimisation - Reduced network bandwidth - Opportunities to improve service availability - Secure by Data Element or set of elements; Simple to redact data	Amazon partial JSON, Microsoft ODATA standard, SQL and Quel all provide this style of request. It provides powerful opportunities for optimisation, hides implementation, and delivers good backward compatibility when combined with tagged requests with assertions. Response schema being defined as a table is critical to the flexibility of this technique if data redaction, extensibility, and field security is to be simple. Use for: EAI, P2P, API, SOA, wAPI (recommended)
Micro-Services	Small services that perform one (small) task that has been identified as useful to a consumer. Examples include getting specific data for a form or drop list, or updating a few fields identified for a specific application. Unless correctly designed, these "services can be troublesome: - Break many of the SOA principles - Lead to proliferation of interfaces - Increase chatter between client and core - Reduces optimisation/backward compatibility - Secure by Operation; difficult to redact data	Commonly used by companies with mobile strategies to reduce 4G traffic, however they tend to lead to literally thousands of services being published. (See Immutable interfaces) Declarative schemas with assertions deliver all the benefits without the costs. Use for: wAPI (not recommended)
Macro-Services	Mega services that perform large tasks based on a fixed schema, for example get all customer data, are inefficient with data movement, marshalling, and collection. They increase latency and particularly when sparse schemas are used, programmer confusion and error rates increase. The large schemas don't do much to support sustainability and while the number of services doesn't increase but confusion and cost of ownership can impose a significant impact. Unless correctly designed these "services can be troublesome: - Break many of the SOA principles - Lead to complex and inefficient interfaces - Increase data between client and core - Reduces optimisation/backward compatibility - Secure by Operation	Commonly used by companies trying to avoid service proliferation or that use SOAP standards. Tend to lead to more work for the provider and client of which we estimate 80% is not useful work because the data is not used. (See Immutable interfaces) Declarative schemas with assertions deliver all the benefits without the costs. Use for: EAI, P2P, API, SOA, wAPI (be careful)