

Transforming BSS/ OSS systems to Microservices Architecture

Authors

Nikhil Mohan, Senior Technology Architect, Infosys Limited.

Ansoo Susan Thomas, Technology Architect, Infosys Limited.

Abstract

Present day telecommunication service providers are under immense pressure to constantly reinvent themselves to adapt their product and service lines to satisfy the modern-day customers, be it residential or large enterprise-scale organizations. Today, IT organizations of almost all telecom players, big and small, make use of smart analytics platforms to make sense of their voluminous and diverse business and operational data. These intelligent solutions help consultants and analysts to draw insights into market behaviour, sales order distributions and other interesting trends and patterns. In this context, there is a need, stronger than ever before, for these telecom companies to focus on optimizing their ‘concept to market’ (C2M) and ‘order to activate’ (O2A) strategies to emerge as clear differentiators in this fiercely competitive industry. C2M and O2A processes rely heavily on the operational agility and architecture robustness of the companies’ BSS/OSS ecosystem. In this whitepaper we discuss the features, merits and methodologies for adopting Microservices Architecture when transforming today’s telecom BSS/OSS IT stacks, equipping them to be ready to take on the volatile and vibrant business needs of tomorrow.

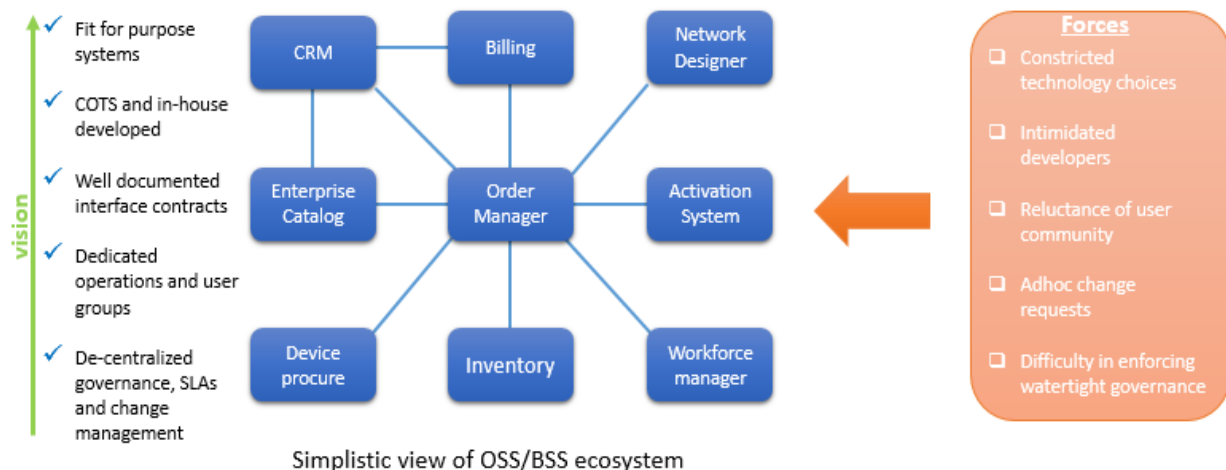
Introduction

In a typical telecom service provider organization, BSS/OSS ecosystem is a complex environment consisting of a plethora of systems. These systems are quite heterogeneous in nature. Some may be COTS products, some custom-built over COTS products, some built in-house and some may be built by third-party vendors. The differences do not end there. Implementation technology could be vastly different. Some built using languages adhering to standard specifications while some maybe using vendor proprietary languages. To add to the mix, there would be applications that are generations apart – one could be a legacy application built using Mainframe while another one could be based on server side java script code running on *node.js* backed by a NoSQL database. To compound matters even further, these systems do not stay the same. Many of the constituent systems evolve enormously over time to cater to the business and IT needs of the organization. As a result, the IT stack, once envisaged by the enterprise architect as a conglomeration of neatly defined IT systems, COTS as well as custom built, integrated using well-documented interface contracts, has evolved into a group of unmanageable tightly coupled monoliths.

The order to activate cycle in a typical OSS/ BSS ecosystem could be explained as follows: A CRM system manages the customer management and product management functions. The CRM interfaces with the Enterprise Catalog to retrieve the details of products those are to be sold through the stack. Inter product association details, business rules, pricing details and relevant product attribute details are also fetched from the catalog. A sales user is a typical user of this system. The CRM creates product orders and passes

them to the Order Manager for performing product to service decomposition, perform service design and generate work orders for work force management system to schedule work allocation. The Order Manager fetches the service and resource specifications from the Enterprise Catalog to perform its actions. As a step in service design, order manager interacts with Network Design System that performs functions such as IP address allocation, network topology definition and other design activities. The Order Manager also acts as the orchestration engine for sending order requests to Procurement system for purchase of bill of materials when required. Once the service and resource configuration steps are completed, the provisioning staff uses Order Manager to invoke the Activation system for activating the service in the network. The applicable billing codes are also sent to the Billing system for sending purchase orders to the customer. As a final step in the fulfilment flow, the Order Manager sends a response to the CRM indicating the order status as completed.

Fulfilment processes for major telecom companies in the real world could be much more complex. There could be more systems involved, comprising of complicated workflows and a much higher magnitude of data flow between systems. But the fundamental principles on which these systems operate remain the same. Any additional complexity only accentuates the need for IT transformation.



As shown in the above figure, a lot of forces act on these IT systems. Over time, the consequences of these forces lead to the degradation of qualities that the enterprise architect and other stakeholders, at the time of inception, had deemed appropriate to ensure the architecture robustness.

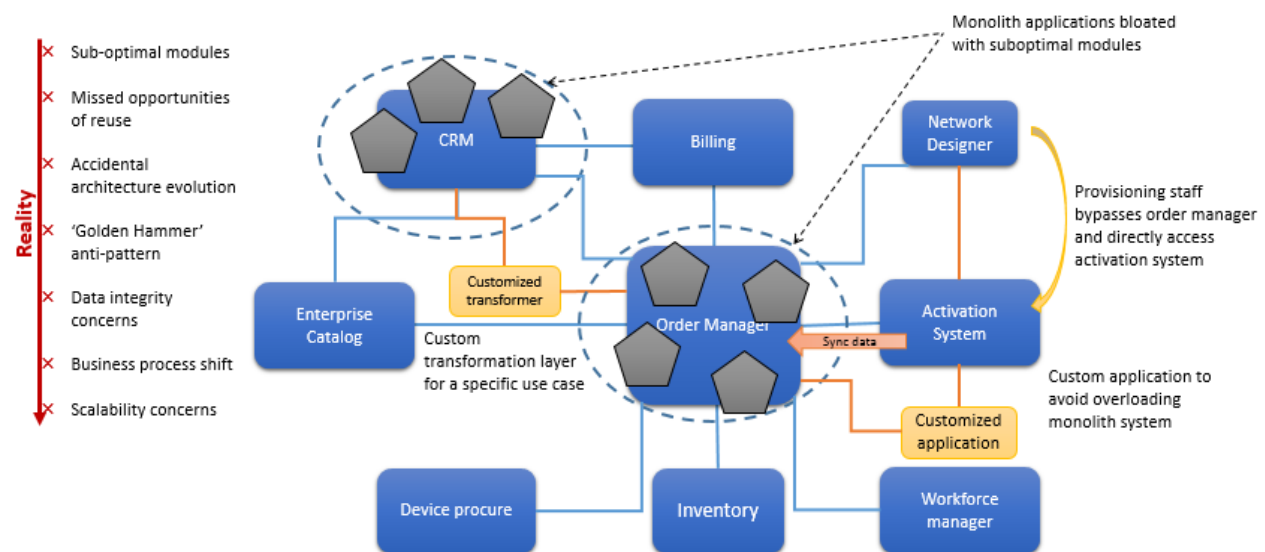
A brief look at these forces would indicate that people or processes drive most of them. But a further glance would show that these are mostly enterprise architecture concerns and not a lot can be attributed to specific people or process issues.

Traditionally, IT systems in large enterprises would be built out of commercial-off-the-shelf (COTS) products. These COTS products offer developers with varying degrees of flexibility to extend the solution for context-specific use. But invariably the technology stack for these products would contain proprietary elements such as source code language, toolkits, configuration elements, operating environments etc. The developers working on these systems have the challenge of working around these numerous invariants to accomplish extensions to the COTS solution. Even in cases where open source technologies are used by the solution, compatibility issues might arise. For example, a new requirement that can be implemented easily using newer versions of a particular open source software cannot be applied to the solution since the operating platform may be quite old. The business will most likely not have the appetite

to spend for a product upgrade that involves hefty license costs. Why fix something that is not broken? In this case, the developer would have to settle for a suboptimal solution by choosing an older version of the software that is compatible with the overall solution.

The users of the system also contribute to the set of forces. Over time, as the applications undergo wear and tear, user frustrations tend to increase. In software terms, wear and tear generally refers to more load, increasing response times, increased usage complexity, more bugs, data correctness issues, increased defect resolution times and increased frequency of system crashes. Even though it is not proper to generalize, operation teams of most of the large enterprise scale systems that have been in production for a few years face these challenges. It is not uncommon for these frustrated users to try and bypass these flawed system components and define an alternate un-written business process by themselves. These may lead to serious ramifications for both business and operations. These alternate business processes might involve manual steps that lengthen the order to activate cycle times. Also, it can increase chances of mistakes resulting in higher workload for operations support. Operations support face the headache of resolving all data integrity issues that could emerge.

As systems tend to become bigger and bigger with new functionalities, the team responsible for supporting and managing the systems also grows. Also, new stakeholders could come in, there could be change requests that require modifications not aligned to the original system architecture. It is not reasonable to expect the same stakeholders who built the architecture to be around for the lifetime of these systems. People move and so should systems. But when systems become large monoliths, impermeable governance becomes extremely difficult. Some changes can slip under the radar.



As shown in the figure, even for a well-managed group of systems, architecture decay occurs due to the various forces that act on these systems. Some of these applications might grow into unmanageable monoliths due to the proliferation of suboptimal modules. Functionalities that need to reside elsewhere might be added to the application. Conversely, due to the complexity of the solution, functionality that needs to fit into the solution might be pushed out. The common pattern emerging from this traditional approach of defining enterprise architecture is rigidity. The lack of flexibility of these systems forces developers to design and implement new functions as enforced by the architecture. As a result development teams could reject proposals for changes to the system due to scalability concerns. The user

community also would be averse to making changes to the system, as they fear it would further deteriorate the system.

In this context, what we need is a transformation strategy that addresses the problem areas of all stakeholders of the OSS/ BSS ecosystem – business, users, operations, developers, architecture group and marketing. A strategy that transforms the enterprise architecture to loosely coupled systems that are easier to build, test, manage, integrate, use and monitor. These agile systems should also have sufficient intelligence built-in to help analytics platforms to quickly retrieve relevant data patterns for smarter decision-making, be it for operations or for business.

Traditional approach for transforming BSS/ OSS architecture to loosely coupled systems

Traditionally, the transformation of enterprise IT stack to loosely coupled systems would point to service-oriented architecture built around a complex middleware system such as an Enterprise Service Bus. To make the architecture complete, a Business Process Management (BPM) system sits above the ESB to compose business processes out of the published coarse-grained services.

This approach follows a layered architecture pattern consisting of three layers. A bottom-up approach would be the best choice for implementing this architecture. The layers are identified by the functionalities they perform:

Service enablement

This layer represents the lowest layer representing the fine-grained services published by the various applications in the BSS/ OSS ecosystem. The first activity of transformation is to identify and extract functional units of each of the applications as services. This activity is referred to as service enablement.

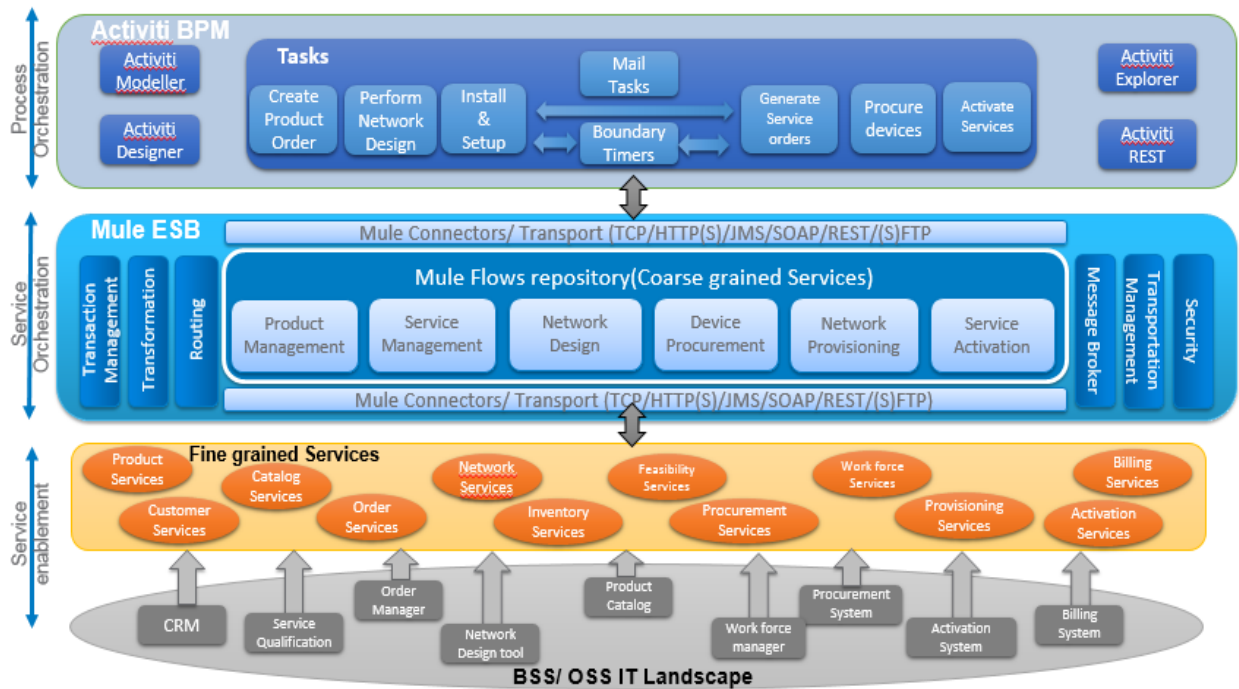
Service orchestration

An Enterprise Service Bus resides in this middle layer that performs orchestration of the fine-grained services developed and published by the lower layer to form coarse-grained services. The functionality achieved in this layer is referred to as *Service orchestration*.

Business Process orchestration

Above the ESB, Business Process Management system resides as the top most layer in the architecture stack. The job of BPM is to compose business processes out of the coarse-grained services published by ESB. BPM helps define business process workflows consisting of manual as well as system tasks. The functionality achieved in this layer is referred to as *Business process orchestration*.

Two popular open source software, Mule by *Mulesoft*, which is a popular open source ESB implementation as well as Activiti, a widely used open source BPM implementation, are put to use for representing the reference architecture for this service-oriented approach. Both Mule and Activiti are widely used in production scale environments.



The technical architecture explained earlier is illustrated in the above diagram. The lowest layer is constructed first. The functional capabilities of each of the systems are extracted as web services in this layer. For example, CRM system, which does customer management and product order management functions, publishes appropriate fine-grained web services. In reality, this may consist of many fine-grained APIs such as *retrieveCustomer*, *retrieveCustomerLocation*, *createProductOrder*, *validateProductOrder* etc.

Similarly, the other applications in the ecosystem also extract and publish appropriate fine-grained APIs. The architecture does not force fit any particular protocol for these services. They can be represented as SOAP based services adhering to WS-* standards or relatively lighter REST services. The fine-grained services published by the various applications need to collaborate and perform a logical unit of work for the corresponding business process. The middle tier performs this service orchestration. This is achieved using mule ESB. Mule supports a host of protocols including and not limited to TCP, HTTP, SMTP, (S) FTP, AMQP, and JMS. Mule flows are used to compose the coarse-grained services. For the BSS/ OSS problem domain, the key mule flows are identified as Product Order Management for creation of product orders, Service Order Management for service order creation, Network Design for network topology creation, Device Procurement for bill of material procurement request, Network Provisioning for device configuration and validation and Service Activation for service activation functions.

Some of the important features of Mule are listed below:

- ✓ Service Creation and hosting: Mule can be used as a service container to expose and create services.
- ✓ Service mediation: Mule does message transformation and enrichment on messages thereby freeing up services from doing these. Services do not have to be aware of the different protocols or formats that would be used to invoke them.
- ✓ Message routing: Mule handles routing, aggregation and filtering of messages. Mule also supports rule-based message routing.

- ✓ Business Activity Monitoring and Complex Event Processing: Mule has a built in Business Analyzer module to keep track of all business events happening during the lifetime of an application. Mule also integrates well with Complex Event Processing engines like Esper.
- ✓ Cloud support: Mule Cloud Hub is an IPaaS (Integration Platform as a Service) offering. It has a robust set of connectors for connecting both on premise and cloud-based applications.

The top layer in the stack comprises of Activiti BPM for orchestrating business processes. BPM helps streamline manual business processes that play an important role in improving the overall operational efficiency. Activiti provides the option to configure boundary timer events that serve as an excellent mechanism for jeopardy management. Activiti BPM supports BPMN 2.0 standard for defining business processes. Activiti comes with a designer component that helps to model business processes intuitively. Activiti Explorer that is bundled with the Activiti installable is a web application that can help with the administration and monitoring of Activiti processes.

Why does the traditional SOA approach not work?

One of the founding principles of Service Oriented Architecture is to build large systems as a collection of loosely coupled services that collaborate over the web. The emergence of ESB paradigm in this space can be attributed to address some of the cross cutting requirements such as security, mediation, monitoring etc. Over the years, attempts to productize ESB has resulted in adding too many features into this middleware leading to highly complex architectures. Subsequently, development projects adopting this architecture have steep learning curves resulting in slower development and release cycles. The ESB layer often ends up having complex business logic built into them for message enrichment, transformation and routing. As more and more applications/services in the enterprise use the ESB for integration, the ESB becomes bulky with a lot of application specific logic. Over time, ESB loses its scalability and becomes more and more difficult to maintain. It becomes a bottleneck and a single point of failure, which can bring down the entire enterprise. Consequently, the ESB emerges as the most critical component in the architecture, leading developers to implement services to align to the middleware rather than the other way around. Services tend to lose their autonomy, the fundamental principle of SOA.

SOA was envisaged to be built around reusable business services. When SOA is implemented using business process orchestration approach, it is likely that a few central services end up handling important business logic that it should not ideally contain. This leads to higher coupling and lower cohesion. In short, in practice, the architecture will end up having a few bulky services doing a lot of things and a few anaemic services doing very little.

Introducing Microservices Architecture

Now let us turn our attention to an architectural style that has generated a lot of interest in recent times. Microservices architecture is essentially about developing an application as a set of small autonomous services that are independently deployable and that communicates among each other using lightweight protocols such as REST. Each microservice executes on its own process and should achieve a single business capability. An important characteristic of Microservices architecture is that there is no central management component for these services. For this reason, Microservices architecture prefers choreography to orchestration for business process realization.

Before attempting to map our problem domain to Microservices architecture, it might be worthwhile to examine the reason why this architecture has generated so much interest in the technology world. As we have seen earlier, the various forces that continuously act on the enterprise architecture eventually make some of the applications in the system huge, bloated and rigid. These are commonly referred to as monolithic applications. These monoliths suffer from obvious drawbacks:

- ✓ The whole monolith needs to be built as a whole – even a small change has to go through a rigorous release and deployment cycle
- ✓ Scaling is expensive – Horizontal scaling can be applied to a monolithic system only in its entirety. Limiting scaling just to the affected module might not be feasible.
- ✓ Locked into a technology stack – It might not be possible to adopt the most suitable technology for a particular functionality resulting in a suboptimal solution
- ✓ Intimidated developers – Codebase size may put off development teams from working with those systems.

These limitations have led to the evolution of Microservices architecture. Since microservices are built around business capabilities they enable y-axis scaling. Also, since microservices are deployed in their own processes, x-axis scaling can also be easily achieved. Since each of microservices are independently deployable, polyglot programming approach can be adopted where the most appropriate technology stack can be chosen for satisfying each business capability.

Applying Microservices architecture to BSS/ OSS landscape

As opposed to the traditional SOA approach, microservices should be designed as fine-grained services communicating over lightweight protocols. So, we prefer REST to the comparatively heavier SOAP-based web services. The business capabilities of systems such as CRM, Order Manager, Network Designer, Enterprise Catalog and other systems in the IT stack are transformed to fine-grained functional units published using Restful API. REST is not the only communication medium for microservices. As we briefly mentioned earlier, choreography is the preferred approach for data flow through the system.

Achieving choreography in microservices

Choreography is a fully distributed approach where each component of the whole system is aware of its job and performs those jobs in reaction to pre-defined event occurrences. Hence, we need asynchronous messaging middleware for implementing choreography. In Microservices world, the backbone for implementing these messaging channels is referred to as Message Bus. The Message Bus is designed in accordance with the Microservices architecture principle of “smart end points and dumb pipes” and hence is much lighter and simpler than the Enterprise Service Bus. *RabbitMQ* based on Advanced Message Queuing Protocol (AMQP) is a popular choice for building the Message Bus. So the microservices that perform an action that implies the completion of a business process task should generate an event that needs to be published to Message Bus. The microservices that have subscribed to this event would start executing once this event occurs. This choreography model ensures that the collaborating microservices are as decoupled from each other as possible.

Containerization of Microservices using Docker

Another critical factor for successful Microservices Architecture implementation is sound DevOps adoption. Infrastructure automation through Continuous Delivery is an important aspect of Microservices Architecture. Docker containers have proven to be a very appropriate platform for deploying microservices. Since Docker runs directly on OS kernel, overheads such as network bandwidth and storage that are associated with hypervisor-based virtualizations are minimized. One major benefit of Docker-

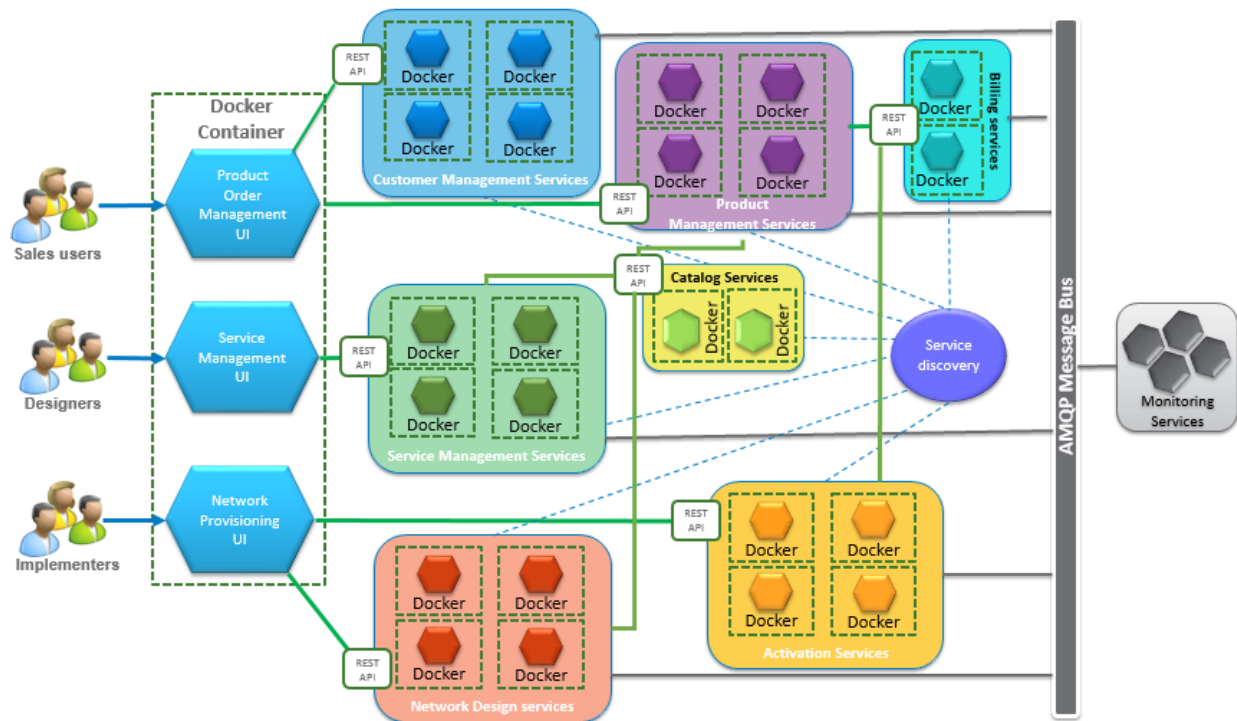
based deployments is the environment consistency it offers. Once Docker image for a microservice is generated using Docker file, the Docker image can be deployed to any environment. The ideal deployment strategy is to deploy a single Microservice instance per Docker container. For achieving horizontal scaling, the same microservice can be deployed on multiple Docker containers. Continuous delivery of Microservices on Docker containers in development, test and production environments is facilitated using configuration management systems such as Puppet.

Service Discovery

In a typical Microservices Architecture, we have many instances of Microservices running on Docker containers in a multitude of operating platforms. The ability to dynamically discover services in such an ecosystem is critical to the success of the Architecture. Service Discovery component is responsible for discovering the services that are online and available for consumption. Service Discovery becomes more important in a highly dynamic environment where services are spun up and torn down quite frequently. Several open source service discovery solutions such as Apache ZooKeeper, etcd, Eureka are available in the market.

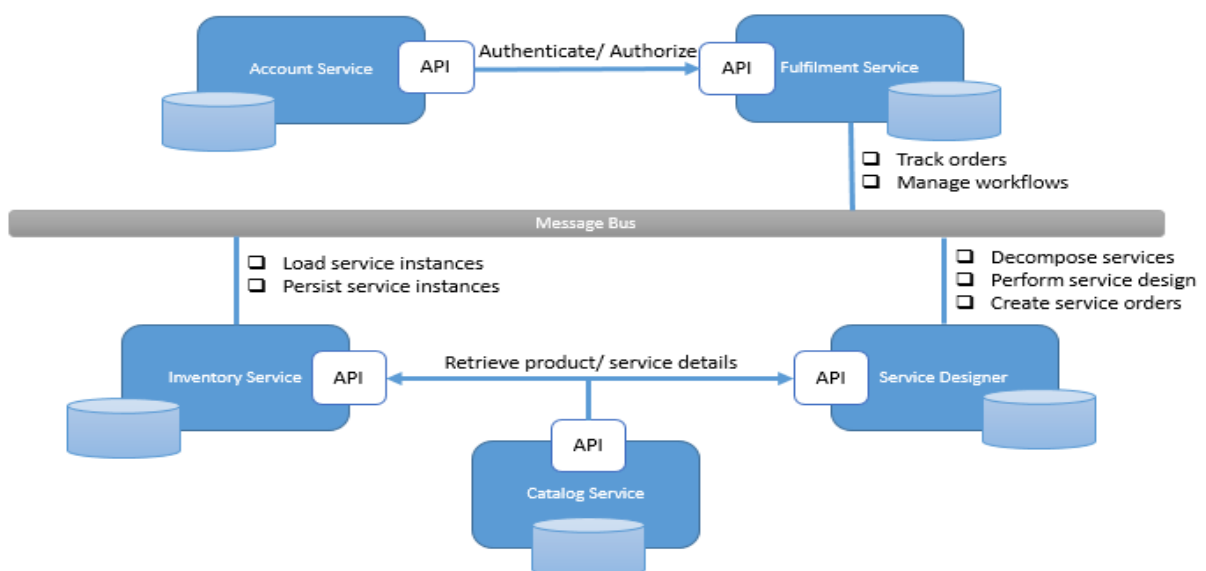
Microservices Architecture for BSS/ OSS ecosystem

Diagram depicting Microservices Architecture for BSS/ OSS ecosystem is given below. For simplicity, all UI services are shown as residing in the same Docker container. This is a design decision to make, considering aspects such as availability, scalability, load sharing, maintenance costs etc. The Architecture also includes monitoring services that subscribe to the Message Bus. Unlike in BPM based approach, Microservices Architecture does not encourage the presence of a centralized orchestration component. But as we have seen, an effective mechanism needs to be in place to track the progress of business process flows. The recommended strategy is to build an overarching monitoring service that subscribes to business process boundary events propagated by the various microservices. This monitoring service preserves the choreography model of the architecture by not orchestrating the business flow, but can have intelligence built in to deal with jeopardy management functions. One idea could be to model Activiti BPM itself as a passive monitoring system, which simply listens and updates the state for relevant events.



Representing a monolithic Service Order Management solution as loosely coupled microservices

A typical Service order management solution will perform service design, order fulfilment and service inventory functions. The following figure illustrates the collaboration among service management related microservices. For this example, microservices for this application has been identified as Account service, Order fulfilment service, Inventory service and Service designer service. Depending upon the complexity of the solution, there could be many more potential microservices. The Service designer and inventory services invoke APIs of Catalog service.



The fulfilment service follows the choreography model for managing the order workflows. The fulfilment service, inventory service and the service designer follow the publisher subscriber model using the message bus to achieve this choreography. This is in line with microservices philosophy of 'smart end points and dumb pipes'.

Key characteristics of Microservices Architecture

In this section, we look at two important characteristics that play a vital role in fulfilling the envisaged goals of Microservices Architecture namely polyglot portfolio adoption and DevOps adoption.

Polyglot programming and polyglot persistence

One of the most important characteristic of Microservices Architecture is polyglot programming and polyglot persistence. Since each microservice follows single responsibility principle and is independently deployed, developers have the freedom to choose the technology stack most appropriate for the given context. Migrating a microservice from a given technology stack to another also should not impact any other service.

The recommended inter-service communication mechanisms are:

- Restful APIs using JSON data format
- Event driven asynchronous communication over a light weight messaging bus (AMQP based message-oriented middleware such as *RabbitMQ*)

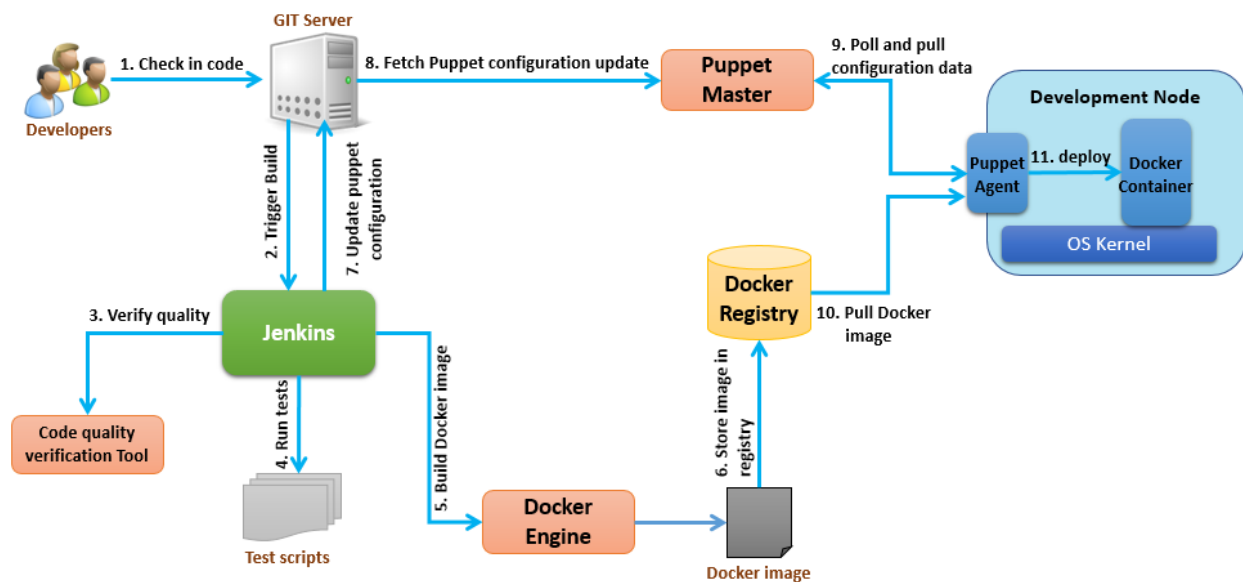
The conventional approach to enterprise application development is to rely on proven technology stacks such as Java/ J2EE or Microsoft based stack for application development and any RDBMS such as Oracle or MySQL for the backend. There are numerous frameworks and best practices that are available for these technology stacks. Microservices development can continue to happen on these technologies seamlessly. Alternatively, microservices can also be developed on any other technology stack that may be more aligned to the problem domain.

Below table lists some of such technology options that can be evaluated for implementing microservices.

Technology	Design considerations
Node.js	NodeJS would be best suited for event driven, IO intensive services. Aids faster development.
Akka	For compute intensive, highly concurrent use cases Akka would be the best fit. Follows the reactive programming paradigm.
Spring Boot	Spring Boot can be used for rapid application development. Spring Boot works well with Docker.
MongoDB	Document database that is best suited for use cases that involve large data retrievals. Can be chosen when better performance is preferred over consistency.
Neo4J	Graph database is best suited for representing data model that is logically aligned as interconnected objects. Hierarchical data model representation for catalog repository is a typical use case.

Adopting DevOps for Microservices

As highlighted earlier, agile development and release are a key feature of microservices. Sound adoption of a robust DevOps framework is essential to achieve this. The diagram below shows an effective DevOps model for Microservices using Jenkins, Docker and Puppet. Jenkins is the continuous integration platform that manages the orchestration of development lifecycle activities. The microservices developer checks in code to a source control system such as GIT that triggers build on Jenkins. On the compiled code, Jenkins runs jobs that generate code quality metrics and run unit and functional tests. Tools such as SonarQube can be very useful for code quality verifications. Once Jenkins establishes that the build is stable, it runs a job to build Docker image using the corresponding Docker file. The generated Docker image is subsequently placed in Docker private registry. Jenkins then proceeds to check in the generated image details in Puppet configuration repository in GIT. A Post merge hook script that is configured to this repository in Puppet master gets triggered on this configuration update, resulting in the configuration files being reloaded to Puppet master. Puppet agents running on the various target hosts on which the microservices are to execute, periodically polls and retrieves the catalog from the Puppet master consisting of node specific deployment instructions. The corresponding agent that receives the relevant instructions then proceeds to execute them thereby deploying the new Docker image to Docker container running on the corresponding host.



DevOps for Microservices using Jenkins, Docker and Puppet

Microservices Architecture Suitability Assessment Model for legacy enterprises





As discussed in the beginning, IT applications in BSS/ OSS world are complex systems. In this context, transformation to Microservices Architecture is not straightforward. Return on Investment would need to be assessed before going ahead with this approach. In this section, Microservices Architecture Suitability

Assessment Model is proposed to enable architects to make a well-informed methodical evaluation of existing application architecture to take a call on whether microservices is the way to go.

To start with, we rank applications in the IT stack by monolithic characteristics. The typical characteristics exhibited by monoliths are:

- ✓ Lengthy start up process
- ✓ Large deployment windows
- ✓ Slow running test suites
- ✓ Lack of options to scale by functional units

Once the applications are ranked, we apply the suitability assessment model. The model determines the degree of suitability of the application to be transformed to microservices using a 'Goals vs. Challenges' graph that has Goals on the y-axis and Challenges on the x-axis. The graph is divided into four quadrants to help the model categorize the applications according to its readiness of moving to Microservices Architecture. This model consists of a set of criteria parameters that would be applied to each of the applications to assess the application's perceived readiness to move to Microservices Architecture. Each criterion may belong to either of the two categories – Goals or Challenges. The criteria belonging to goals would be assessed in terms of 'Extremely Important', 'Moderately Important', 'Less Important' and 'Unimportant' in the decreasing order of business value. Similarly, the criteria belonging to challenges would be assessed in terms of 'Trivial', 'Easy', 'Medium' and 'Difficult' in the increasing order of complexity. The application subject matter expert completes the assessment for a particular application and applies the report to the assessment model. The model applies the following rule to determine on which quadrant the concerned application belongs to.

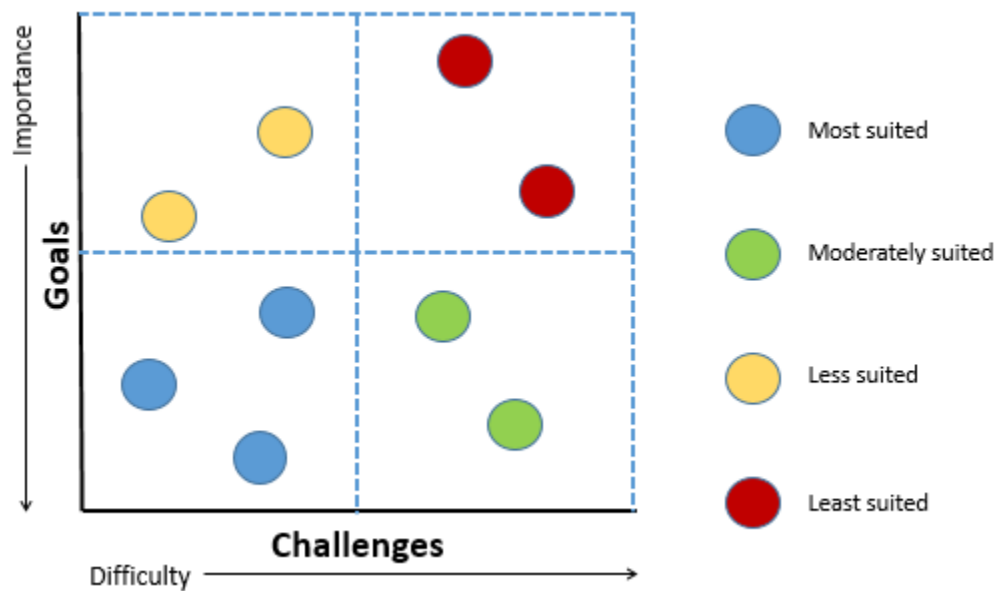
Legend	Quadrant	Goals Measure	Challenges Measure
	Quadrant I	More than half of the criteria fall in 'Extremely important' or 'Moderately Important'	More than half of the criteria fall in 'Easy' or 'Trivial'
	Quadrant II	More than half of the criteria fall in 'Extremely important' or 'Moderately Important'	More than half of the criteria fall in 'Medium' or 'Difficult'
	Quadrant III	More than half of the criteria fall in 'Less important' or 'Unimportant'	More than half of the criteria fall in 'Medium' or 'Difficult'
	Quadrant IV	More than half of the criteria fall in 'Less important' or 'Unimportant'	More than half of the criteria fall in 'Easy' or 'Trivial'

A sample assessment model for a classic enterprise application is given below for reference. This criteria set is not all-encompassing and should be extended or pruned as appropriate for the enterprise context.

Goals based Criteria set					
Goals	Criteria	Value	Goals	Criteria	Value
Shorter development cycles			Phasing out legacy technology		
	Application initialization time	EXTREMELY IMPORTANT		Out dated technology support	MODERATELY IMPORTANT
	Code build & packaging	EXTREMELY IMPORTANT		Availability of skilled resources	MODERATELY IMPORTANT
	Codebase size	EXTREMELY IMPORTANT		golden hammer' anti-pattern	MODERATELY IMPORTANT
	Inter module dependency	MODERATELY IMPORTANT	Avoid vendor lock-ins		
	Isolated testing readiness	MODERATELY IMPORTANT		Use of proprietary software	EXTREMELY IMPORTANT
	Software currency rigidity	EXTREMELY IMPORTANT		upgrade costs	EXTREMELY IMPORTANT
Continuous Deployment				operational costs	MODERATELY IMPORTANT
	Deployment outage window	EXTREMELY IMPORTANT	Simpler change management		
	Deployment readiness process	MODERATELY IMPORTANT		Change approval process	MODERATELY IMPORTANT
	Build and release process	MODERATELY IMPORTANT		complexity in interface contracts management	EXTREMELY IMPORTANT
Functional scaling			Infrastructure automation		
	Disparity in response times by transaction	EXTREMELY IMPORTANT		extend of automation in deployment	EXTREMELY IMPORTANT
	Transaction based NFR	EXTREMELY IMPORTANT		CI tools adoption	EXTREMELY IMPORTANT
	Resource intensive API usage	MODERATELY IMPORTANT		build status monitoring and remediation process	MODERATELY IMPORTANT
	Query execution time	MODERATELY IMPORTANT	Moving to cloud		
				on demand scalability need	RELATIVELY UNIMPORTANT
				Total cost of ownership	MODERATELY IMPORTANT

Challenges based Criteria set					
Challenges	Criteria	Value	Challenges	Criteria	Value
Coupling			Transaction Management		
	Extend of coupling in code	MEDIUM		Extend & criticality of partial failure scenarios	MEDIUM
	Existence of non standard interfaces	EASY		Acceptance of eventual consistency	MEDIUM
	Extend of separation of concerns	MEDIUM	Security		
	Extend of cross cutting concerns	DIFFICULT		Extend of understanding of threat levels of various functions	DIFFICULT
	public API usage	MEDIUM	COTS monolith		
Splitting Database				Source code ownership	MEDIUM
	Referential integrity constraints	DIFFICULT		Deployment infrastructure ownership	DIFFICULT
	Extend of object relational representation	DIFFICULT		License model	DIFFICULT
Latency				COTS product roadmap and pricing model for Microservices	DIFFICULT
	network reliability	EASY	infrastructure proliferation		
	tolerance on performance related NFRs	MEDIUM		Readiness to move to cloud	DIFFICULT
Testing				Infrastructure support and maintenance process	MEDIUM
	Extend of test automation - unit tests, acceptance tests and integration tests	MEDIUM		failover and disaster recovery process	MEDIUM
	Extend of proactive application monitoring	MEDIUM	Service versioning		
bounded context determination				Source control management practices	MEDIUM
	Degree of coupling and cohesion	MEDIUM		Extend of modularization in source code organization	MEDIUM
	Domain knowledge	EASY			
	Extend of traceability of business processes to code	MEDIUM			

The evaluation rule used by the model is as follows:



The Suitability Assessment Model provides an excellent reference point for evaluating the readiness of the enterprise architecture to move to microservices.

Methodologies for transforming legacy BSS/ OSS enterprises to Microservices

In this section, we look at a few methodologies for transforming the applications of the IT stack to Microservices Architecture keeping the Suitable Assessment Model as the basis.

When to prefer centralized orchestration over Microservices

As a rule of thumb, if three of the five top-ranked monolithic applications in the IT stack fall in quadrant III, then transformation to Microservices Architecture might be difficult to achieve. It might be better to go with the centralized SOA approach; the ESB and BPM based one. Even though we recommend Microservices architecture as the strategic route to take, this is one scenario where it might be wiser to go with the centralized orchestration approach. Having loosely coupled SOA is better than tightly coupled point-to-point connected architectures anyway.

Methodology 1 – Transformation of ‘most suited’ applications

Migration of applications in quadrant I to microservices offer the highest value in the transformation exercise. These applications are classified as ‘most suited’ because the value offered by their transformation to Microservices Architecture is of highest importance to organization’s business goals as

well as they offer least challenges for migration to Microservices Architecture. The following sequence of steps indicates the recommended methodology for achieving the desired transformation for these applications.

Identify bounded contexts

The idea of bounded contexts is drawn out of domain driven design which means that any given domain consists of a collection of bounded contexts, each consisting of models some of which are completely internal and some of which are shared among external bounded contexts. This notion is aligned with the concept of splitting functional units by business capabilities, which is what microservices are about. So this is the first logical step to be performed in any migration to micro services exercise. Aspects like source code availability and access, traceability of business domain to application code are critical success factors for this step.

Organize source code to appropriate domains

Once we have identified the bounded context, the next activity would be to remap the source code to the corresponding domains. In the case of Java applications, the concept of packages can be used to map the source code to its corresponding domains.

Analyse Data access layer

Now we look at data access layer code to identify database tables that are close to each of the domains. This step is important because, in microservices world, we need to keep the data store private for the particular microservice.

Identify corresponding tables and relationships

Since most monolithic applications traditionally use RDBMS to manage its persistence layer, this step is normally quite involved. Here the relationships between tables are analysed to identify the database coupling.

Use API calls to avoid foreign key references

Now our aim is to look at design options for removing the foreign key references so that we can achieve data store isolation. This is important since we need to keep the associated data for a service confined to that service. One approach would be to move the relationship from the database to the service layer. The referential data would be published using API by the provider service. For example, if Product Offering service in CRM solution space needs the corresponding *product specification id* from the product catalog service, then instead of mapping this relationship in the database, product catalog service would provide the *product specification id* using published API.

Derive new bounded contexts from shared data

In large monolithic applications, a common pattern that is seen is the use of a particular class of data such as represented in a table or a group of tables being accessed liberally across domains. In this step, this class of data is identified and mapped as a new bounded context. This is an effective process of decomposing macro services to Microservices. Like we saw in the previous step, to isolate the data store for this new bounded context, the accessor methods of relevance are published as APIs for the collaborating services to consume.

Split tables representing data for multiple contexts

In this step, identify those tables that represent part of the data for one domain, another part for a different domain and so on. Such tables can be split into new tables each of which represents the corresponding context.

Define transaction management strategy

One of the major challenges that are encountered in localizing data stores is transaction management. Traditional applications are generally designed to make liberal use of ACID properties supported by RDBMS. In the microservices world, a single transaction would involve API calls across many services. Many of these calls would result in writes to multiple databases. In cases of partial failure, rolling the data back in all the involved microservices is a non-trivial task. The solution architect should make an informed decision whether eventual consistency is acceptable for such cases. Where this approach is fine, an asynchronous retry or error event can be triggered where the subscriber service on receiving this event, either retries the operation or effects the appropriate changes to roll back to a stable state. An alternate approach is to implement distributed transaction management using two-phase commit protocol. In the first phase, all participating services vote back to transaction management service indicating whether their operation is successful or not. Once transaction management service receives positive responses from all the participants, it issues a commit request to all of them. This approach is cumbersome and has many potential pitfalls. The recommended option for transaction management is eventual consistency. In situations where eventual consistency is not enough and ACID compliance is an absolute must, the recommendation would be to avoid database splitting. Another option is to handle transactions at the service level. This is not a cheap option and can add to service layer complexity. At the end of this step, the team should have answers to questions such as following - Which all services can have private databases? Which all services need access to a shared database? What all tables form part of the shared database?

Split database

Now, the bounded contexts that we have identified in step 1 along with the newer bounded contexts that we have derived subsequently have logically become more cohesive. But the code should not yet be split physically. Based on the information gathered from the previous step, we are ready to split the database. This is a good time to analyse whether the data model representation is appropriate. For example, if the data model were representing hierarchical relationships then a graph DB such as Neo4J would be more suited than an RDBMS. At the end of this step, we would have a single monolithic application layer that is logically divided into bounded contexts at design time, consisting of independent database schemas each representing data for the corresponding bounded context. If a decision had been made to change the database technology for a service, then the data model as well as the data access layer of that service also needs to be modified in this step.

Fork new source code repositories

The application is passed through a test suite of elementary test cases to ensure that the architecture is not broken badly at this stage. This round of testing helps to alleviate the blurred boundaries of bounded contexts. After testing is complete, next step is to split the code base. An important feature of microservices is to be independently deployable. For this to happen, it should be possible to build and package the services independently. Splitting the build logic alone is not sufficient. The code repository for each service has to be isolated to avoid chances of tight coupling between collaborating services. Splitting the code base by microservices also helps development teams to manage changes better. The test scripts associated with the domain should also be moved accordingly.

Define service contracts

This step involves identifying the public APIs for each of the services. Here much care should be taken to ensure that the internal service details are as much abstracted as possible. Designers should take extra care to ensure that the service contract is not too convoluted leading to tight coupling. Too many service calls would lead to service orchestration. The design goal should be to achieve event-based choreography through the message bus. The knowledge about one service within another service should be as limited as possible.

Apply transaction management strategy

Transaction management strategy has been defined in an earlier step. As an outcome, the monolithic database has been split into multiple schemas. In some cases, it could be a new database altogether or it could be a new schema in the existing database. In this step, we apply those defined strategies by modifying the service layers to handle transactions across the databases.

Manage security

At this stage, the single monolithic application has been split into multiple microservices. From a security perspective, the complexity of the architecture has increased. Monolithic applications would typically have a single entry point where the user is authenticated and authorized. Once the user is authorized, the user would be free to access the backend services accordingly. This approach is impractical and inefficient for securing microservices. Microservices are not designed to maintain user principal information across API invocations. This information can be passed around in the form of access tokens, cookies or session attributes. In addition, microservices should also take care of inter-service authentication.

A few popular and open standards-based methods to implement security in microservices are given below:

- OAuth 2.0
- JSON Web Tokens (JWT)
- OpenID Connect which is built on top of the OAuth Protocol
- SAML
- Two way TLS

Handle reporting

Typically, in large enterprise applications, secondary database is used for generating large reports. This is done so as to limit the impact to primary database due to long running reports. Also the schema in this database is organized to facilitate quick retrieval of reporting data. The data from primary database will be periodically synchronized to secondary database to ensure consistency. It is advisable to leave the schema in reporting database unchanged. Since schema in both the databases is different, the replication logic should change. Suitable approach is to implement an event pump in each service that listens for a change of data in its domain in primary database and subsequently propagate the change to the secondary database. Even if the application does not maintain a separate schema for reporting, it might be worthwhile to consider introducing one as part of the transformation program.

Containerization of microservices

This is an important step where we split the deployment model for the services. In this step, Docker files for the services are created and are used to generate the corresponding Docker images. Docker images

ensure the deployment readiness of the services on to the various environments such as development, test and production. Best practice would be to maintain a single service instance within a Docker container.

Plan for deployment

QoS (Quality of Service) parameters for the various services are independently assessed at this stage. For services that have high availability, performance and scalability demands, multiple instances would need to be packaged as separate Docker containers that would be deployed on multiple hosts. Container cluster management solutions such as Kubernetes can be used with Docker for this purpose. During this stage, capacity planning is carried out to assess the need for features such as auto-scaling, load balancing, database sharding and caching. Since Docker images run directly on operating systems kernel, the deployment infrastructure is abstracted from the service layer enabling service instances to be deployed to cloud computing platforms such as Amazon EC2 or Microsoft Azure or on premise infrastructure. A suitable service discovery solution such as etcd or Apache Zookeeper also needs to be set up to complete the deployment planning.

Setup DevOps for continuous delivery

Now that we have completed the basic setup for migration to microservices, the next step is to set up a streamlined DevOps process. This includes adapting continuous integration solution such as Jenkins to automate code analysis, build, unit test and functional tests for the microservices. Configuration management systems such as Puppet can be used to perform continuous delivery by automating Docker-based deployments. This automation helps deploy microservices to multiple environments without manual intervention. Test automation scripts may need to be altered to align to the microservices scope. An efficient DevOps framework is a must for achieving the agile development and release cycles that are a fundamental characteristic of microservices.

This concludes the sequence of steps that need to be followed for migrating the ‘Most suited’ applications to Microservices Architecture.

Methodology 2 – Transformation of ‘moderately suited’ applications

The set of applications that fall under this category are most critical to the transformation project. Like the applications in quadrant I, their transformation to Microservices Architecture are of highest value to the organization’s business goals. But contrary to ‘Most suited’ applications, the challenges involved in the migration of these applications are quite difficult to overcome. Hence, project team should take maximum care in executing the transformation of these applications.

The steps involved in methodology 1 are comprehensive, but not all of them can be applied in methodology 2. In this section are listed the steps that are difficult to achieve for this group of applications. Alternate strategies to be adopted for these steps are also discussed.

Organize source code to appropriate domains

Constraints such as lack of availability of source code due to licensing issues, lack of support for a legacy product etc. hinder the flexibility in performing this step. One option could be to replace the legacy product altogether, but this might not be practical due to various factors. Alternately, the code needs to be organized by domain to the maximum extent possible.

Define and apply transaction management strategy

These steps could be quite complicated for such applications. For use cases that are heavily dependent on the ACID properties of the underlying database, it is advisable to avoid splitting the database for such transactions. Isolate these transactions and proceed with the database splitting steps for the rest of the data.

The two steps mentioned above are most likely to appear as high difficulty challenges for applications in this category, but depending on the enterprise context, there could be other scenarios that appear as difficult challenges. But the general idea is to split the application as granularly as possible by business capabilities.

Methodology 3 – Transformation of ‘less suited’ applications

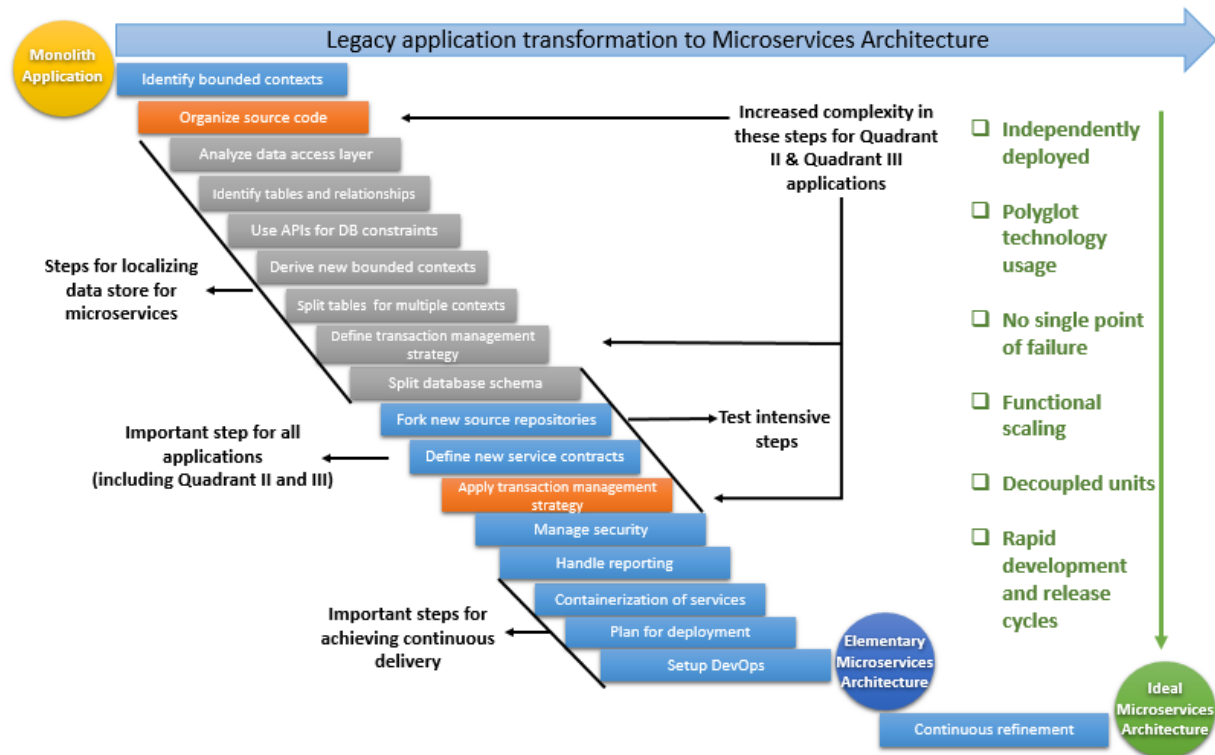
These applications belong to the quadrant IV that is characterized by low importance and relatively lesser challenges. The fact that these applications fall under ‘easier’ challenges category makes them eligible for all the steps mentioned in methodology 1. But, since the transformation of these applications is not important from business goals perspective, either of the following two approaches can be taken:

- ✓ *Transform them first* - Consider these as pilot applications for transformation project. Since the challenges involved are mild they can be easy to adopt for a team new to Microservices Architecture. Similarly, since the goals are not critical, it provides some leeway if things go wrong.
- ✓ *Create Service wrappers* - Since these applications are not strategically important, do bare minimum work on them. Create coarse-grained service wrappers so that they can collaborate in a microservices ecosystem.

Methodology 4 – Transformation of ‘least suited’ applications

As the name suggests these applications are not suitable for following Microservices Architecture. But to make the whole ecosystem work, they need to collaborate with the rest of the services when needed. Like in the previous case, one of the following two approaches can be taken:

- ✓ *Decommission them* - Phase out these applications, as they are not aligned to strategic architecture road map. Revisit their functions and evaluate if they are really required. Very often legacy systems reside in the IT stack for a very long time doing things that do not serve any useful purpose.
- ✓ *Create Service wrappers* - Since these applications are not strategically important, do bare minimum work on them. Create coarse-grained service wrappers so that they can collaborate in a microservices ecosystem. There is not much value in spending effort in splitting the database for these applications.



This concludes our discussion on the various methodologies for transforming different types of applications to Microservices Architecture. The methodologies defined above help to technically transform monolithic applications to a bunch of microservices. But to derive the true benefits of Microservices Architecture, work still has to be done. Code within the services needs to be refactored to improve cohesion and reduce coupling. Microservices need to be inherently designed for failure. Also, an important benefit offered by microservices is their ability to scale independently of the rest of the system. Microservices derived by splitting a monolithic application may not have these characteristics. Also, for the sake of simplicity, changing service implementation technology was not discussed as part of the methodologies. Most teams would not prefer to bring in a lot of disruption mid-way through a transformation project. Once we have a working Microservices Architecture in place, it might be worthwhile to evaluate this aspect and take an appropriate call. A continuous evolution of microservices by applying best practices is important in the transformation journey of an enterprise.

Conclusion

The introductory part of this whitepaper attempts to unearth some of the major pain points encountered in BSS/ OSS ecosystem of typical telecom enterprises. The paper then moves on to define a microservices-based transformation strategy that intends to help organizations adapt and evolve their IT stack. The paper also explores some of the key features and design principles of Microservices Architecture that are fundamental to making the transformation program successful. The paper proposes an open source adoption approach to develop microservices using a reference architecture. The paper also defines a Suitability Assessment Model for assessing the readiness of enterprise applications to be migrated to microservices. Finally, the paper defines various methodologies for transforming applications in the legacy stack to Microservices Architecture by making use of the Suitability Assessment Model as a basis.