|  |  |
| --- | --- |
| ] | **Overview**  Architecture patterns are a general, reusable, solution to a commonly occurring problem within a given context. This section describes the architecture patterns and, more specifically, the patterns for integration. These patterns can be used by solution  architects as building blocks within the solution architectures.  [**About Integration Patterns**](#_qsh70q)  [Principles](#_lbbygsldvdbt)  [**1. Patterns and Messaging Styles**](#_k57ei4sj2l6y)  [Layered Architecture: API Categories](#_l57lxwyck259)  [Categories](#_fvqet3sfc1nj)  [Consumption Patterns](#_v2qovhnazmhq)  [Basic Flow Pattern](#_b2i4bhskkk8m)  [Task](#_ahun9xctm06t)  [Other Patterns](#_85fa1ythzja0)  [Anti-Corruption Layer](#_y919kncxpks4)  [Vendor APIs / Third-Party APIs](#_l3f4eg12wxfm)  [**2. Role of Microservice Architecture in an API-enabled Enterprise**](#_t2esfr88kbl6)  [What is a Microservice?](#_wroor2rbwlu2)  [Characteristics of microservices](#_xcrqvahwqo15)  [What are the benefits of microservices?](#_6mhryq1aytkf)  [Coupling and Cohesion](#_21e6vshhgz6q)  [Orchestration vs Choreography](#_bp6jrhf10szy)  [Service Orchestration - Centralization](#_2p2csry)  [Choreography – Decentralization](#_147n2zr)  [Aggregates](#_sx2qhruh0ujy)  [Microservice vs Graph](#_k4pjmdz7oeiy)  [Event Driven Architecture](#_3o7alnk)  [TM Forum Event and Flow API](#_1hmsyys)  [Eventual Consistency](#_983jvy2bzrnm)  [Moving on From Enterprise Service Bus](#_8jprfi63agax)  [**3. Transitional Architectures**](#_2grqrue)  [Principles applied in Transitional Architectures](#_3fwokq0)  [Transition Strategies](#_1v1yuxt)  [What is a Transitional Architecture](#_dovlxe1dtny0)  [The Strangler Pattern: Incremental Migration](#_da9a9tp3ker7)  [Benefits of the Strangler Pattern](#_cmt99mlj84bn)  [Migration Steps Using the Strangler Patterns](#_2rn05wl8dy16)  [Coexistence Strategies](#_zdkehvm5qgcb)  [Other references for Coexistence Strategies](#_1lk05nl074uf)  [Legacy Wrapper](#_m3wqq3b2khkh)  [Multi-Channel Endpoint](#_m0xxwho9zbxf)  [The Role of an Enterprise Service Bus in the transitional architectures](#_3tbugp1) |

# About Integration Patterns

## Principles

The following integration principles are applied via patterns to reduce integration dependencies of components and move towards zero-touch integration.

These are derived from the general [Reference Architecture Principles](https://docs.google.com/document/d/1gGbQ_Ea0zLxEE0kOLaVz7z0IW9zO--GpKkqvJppbcvc/view#bookmark=kix.7kdg27iecjjh).

|  |  |
| --- | --- |
| **System coupling** | Integrated systems should always **minimize their dependencies** so each can **evolve independently** with minimal impact to other applications.  When applications are tightly coupled they are required to make assumptions about how other systems work: about their data, processes, and applications. When other systems change and those assumptions are no longer valid the integration between systems must be changed. Changes to existing applications may be necessary to enable integration however the new code should enable reduction in future integration code.  **The interface for integrating systems should be specific enough to implement useful functionality, but general enough to allow that implementation to change as needed.** |
| **Integration simplicity** | Strive for single integration points between domains to provide simple integration internally and externally to a domain. Simple integration provides for greater and more secure access control. |
| **Integration technology** | Simplify and strengthen the integration technology landscape by choosing software and hardware that   * do not increase the number of new tools developers are required to learn to integrate applications; ***but also*** * support vendor diversity and ability to change tools when required |
| **Data format** | Adopt standard data and message extension formats to enable the exchange of data between applications, which can be applied as data evolves.  Standard data and messaging formats become the language of data exchange and integration, and minimize the need to have a translator and the associated integration code required. |
| **Data timeliness** | Integration should minimize the time to share data required by other applications. Data should be exchanged frequently, in small pieces, rather than less often, in large unrelated sets.  Applications concerned with the data should be informed as soon as the data is ready to be consumed by other applications. The longer the latency in sharing data, the greater the potential the data will be stale, and the more complex the integration becomes. |
| **Data and functionality** | Applications should not invoke another application’s functionality remotely, as it increases the complexity of integration and the dependency of applications.  For example, two applications which are part of two different bounded contexts should not directly interact; but rather exchange data via the APIs of their respective bounded context. |
| **Asynchronous Processing** | Processing is independent of the sessions on which requests are sent and replies are received. No direct correlation can be made between a request and a reply, and no assumptions can be made about the timing of the reply.  An asynchronous operation is non-blocking and only initiates the operation. Asynchronous message passing allows more parallelism. |

# 1. Patterns and Messaging Styles

* Overall architecture design patterns & constructs to support API planning & development
  + [**Layered architecture patterns enabled by API categories**](#kix.b4pz4jpccsfz) *supporting the principles of loose coupling and separation of concerns*
  + [**Anti-corruption layer**](#dnmjv1v8wunf) *supporting the principle of abstraction*
  + [**Microservice Architecture**](#8hd8673guhn), *providing autonomy, agility, resiliency and enabling the full value of Dev Ops*
    - CQRS - Separation of query and management operations to enable performance and distribution patterns
    - Event-driven architecture *enables loose coupling, autonomy, and separation of concerns*
    - Moving on from the Enterprise Service Bus
  + API Gateway Patterns *providing for secure communication internally and externally*
* Evolving existing systems to the target: [**Transitional Architectures**](#k4hscgbssxq0)

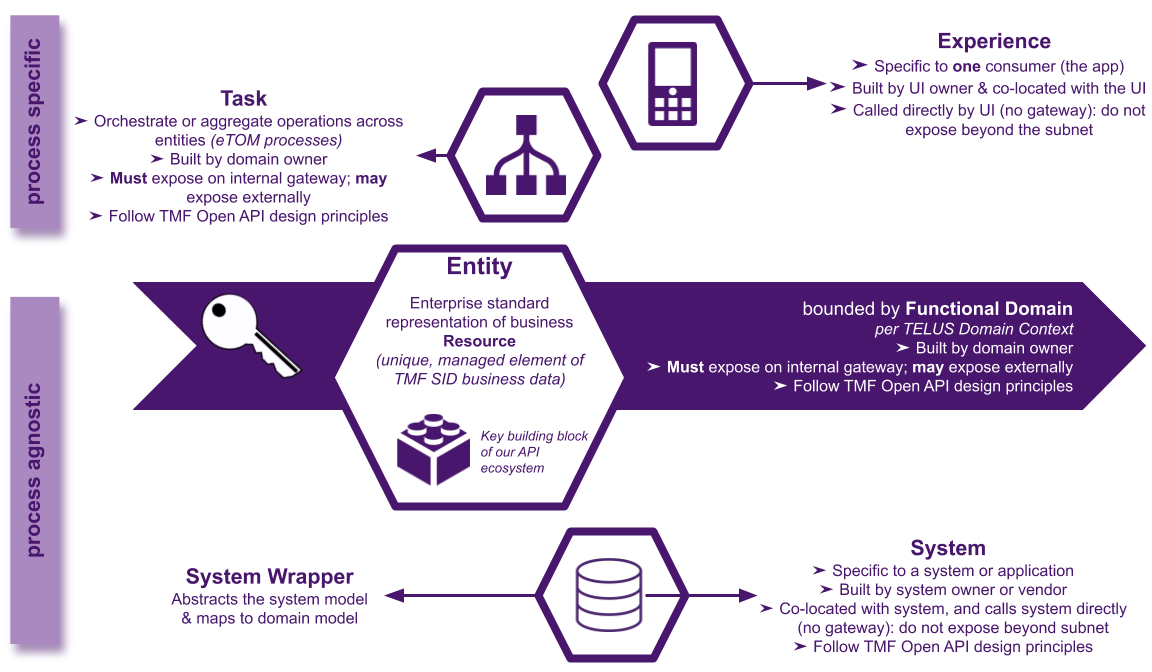
## Layered Architecture: API Categories

API categories guide the realization of a layered architecture by providing a framework for API design and development, ensuring API design conforms to overall Integration Architecture principles:

* Separation of Concerns
* Abstraction
* Loose coupling
* High cohesion
* Single responsibility
* Autonomy *(specifically, by enabling autonomous interfaces)*

Key to the concept of API layers is that API **resources** express Business Entities: they are representations of the entities managed by the application. Operations on resources affect the state of the corresponding entities managed by the API.

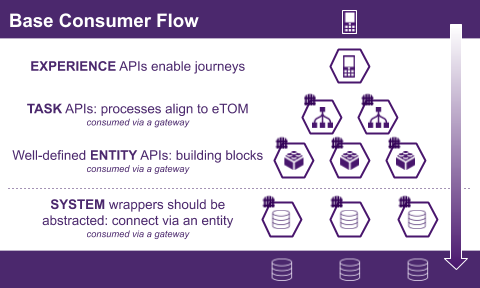
### Categories



|  |  |
| --- | --- |
| **Entity** *key building blocks* | Aggregates and/or mediates data within a **functional domain** to provide a standard representation of a **business entity**. Provides the [**anti-corruption**](#dnmjv1v8wunf) layer *(protects applications from direct interaction with applications outside the domain)*.  **An understanding of** [**Domain Driven Design**](https://docs.google.com/document/d/1H8RyV1G8IByUV6FcJAFrmeRvJQqfJr2YKKbjht7wODo/view#bookmark=id.fq0493ehmo1a)**, and the** [**TELUS domain context**](https://docs.google.com/document/u/0/d/1yUNdmekHoBKbntIFZzgmP3Qvkj6NyLsxudZOmM1lhgI/edit)**, is foundational to mapping API entities and therefore resources which are within the scope of an API.**  [TMF Open APIs](https://docs.google.com/document/u/0/d/1OtMJUWaTH_xnsrQlbr3ZxVmKnOffrAOf5p6xCV0lxOM/edit) are the standard suite of APIs we have adopted to enable our system of core business APIs. They are REST realization of the TM Forum SID entities - they add behaviour to SID by defining operations that can be executed on the entities. |
| **System Wrapper** | Maps between the **domain model** and the TMF **Industry standard model**, and can be used to wrap a System API to abstract from the downstream technology. |
| **System** | Interface to a specific data store or application. |
| **Task** | Performs or plays a part in *(potentially a number of)* business processes from TMF-BPF (eTOM). Can expose complex operations which are not easily decomposable to CRUD Entity-based operations. Can be event-driven.  Response is composed of Entities whose attributes *(as per Entity APIs)* are normally derived from the Information Framework (SID) and/or TMF Open APIs.  Solution option for API responses required to:   1. Orchestrate, or 2. Aggregate, or 3. Create or Execute Task resources   ...for entities, APIs, or microservices within the same or multiple domain (and/or sub-domain) contexts. |
| **Experience** | Orchestrates the activities to support a customer journey within a specific modality or channel. |

### Consumption Patterns

#### Basic Flow Pattern



#### Task

Task API Patterns:

1. Orchestration of (managed or non-managed) resource(s) and/or APIs using a verb that is not easily decomposable to CRUD methods.

2. Microservices Aggregate:

- A design pattern to build software composed of multiple re-useable services that interacts with each other to provide a business capability

- Persistent data for the API is private to the API.

- Aggregation of entities and/or APIs spanning one to multiple domain (and/or sub-domain) contexts

3. Task resource : executable function against one or multiple entities and/or APIs that are necessary when the required action cannot be mapped to standard CRUD methods.

### Other Patterns

* [integration patterns](https://docs.google.com/presentation/d/1zpTH6fijzJe5F1BB1mcW7PHieECBBNTCgTvPomlzVv4/edit#slide=id.g4e30aa7e2c_0_0) (some with layering / domain context)
  + basic inter-layer communication (experience -> task -> entity -> system wrapper -> system
  + within a domain, and between domains
  + external and internal gateways
  + 3rd-party APIs
  + cross-cloud
  + legacy
  + synchronous / asynchronous
    - orchestration
    - choreography
    - events

## Anti-Corruption Layer

Although the platform and domain viewpoints abstract the applications and provide an anti-corruption layer (a layer to protect applications from direct interaction from outside the domain) it is still worthwhile to consider the adoption of good architecture patterns. Although each bounded-context has its own ubiquitous language, as new applications are implemented or integrations are refactored, the standard language of the Open APIs can be adopted to improve the ease of integration and interoperability within a platform. This can be helpful in a large enterprise like TELUS where development teams are currently organized by lines of business or by legacy organizations and are custodians of applications that need to integrate in order to exchange data or integrate in order to provide a complete value stream.

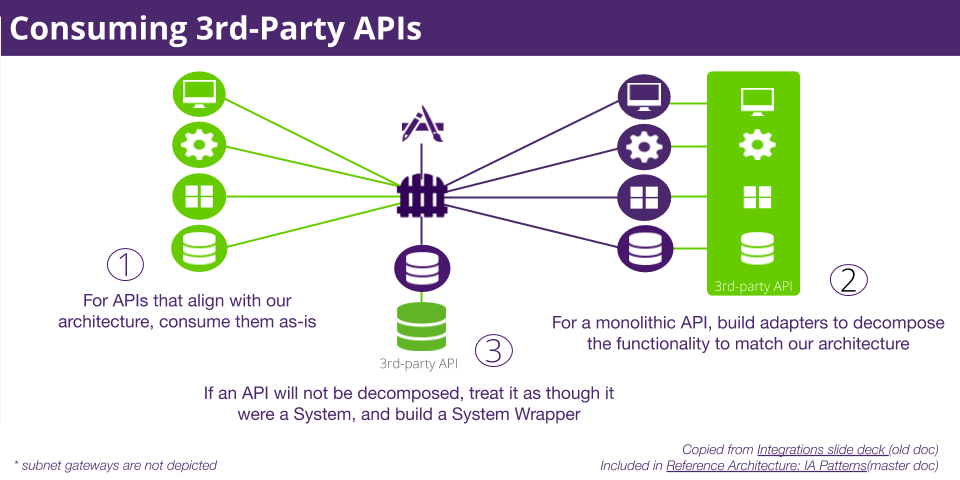
The architecture patterns applied to the target and transitional architectures are industry proven patterns such as the following:

* + Layered architecture
  + Event-Driven architecture
  + Microservice architecture
  + Security

All the architecture patterns are detailed in the Integration patterns document.

## Vendor APIs / Third-Party APIs

|  |  |
| --- | --- |
| **Problem** | TELUS makes use of many 3rd-party systems, and these 3rd-parties may not necessarily conform to our enterprise architecture ideals. |
| **Pattern** | For 3rd-party APIs, our preferred consuming patterns in order of greatest preference are as follows:   1. If their APIs align with our architecture, consume them as-is 2. If they provide one monolithic API that can be decomposed to fit our architecture, build wrappers for their APIs functionality to match our architecture 3. As a last resort, if their API cannot be decomposed (due to nature of the API or lack of resources), use their API as though it were a System and build a system wrapper API   Note that another important part of our architecture is domain conformance. APIs should be decomposed with consideration towards the domains. |
| **When to use it** | Any time you need to consume a service or API provided by a third-party. |



# 2. Role of Microservice Architecture in an API-enabled Enterprise

## What is a Microservice?

Microservice Architecture plays a role in enabling many of the benefits of the target architecture. The frameworks of the target architecture being domain-driven, platform centric with well-defined bounded-context can be implemented through the application of a microservice architecture.

Microservices are “loosely coupled [services] with bounded contexts.” Microservices are an architectural style that functionally decomposes an application into a set of services that use lightweight protocols (HTTP) and have well-defined domain boundaries that focus on single functions. The ability to functionally decompose an application makes microservices an important pattern in the transitional architectures as well as the target.

### 

### Characteristics of microservices

* Single purpose (bounded-context)
* Modular & independent
* Simple, well-defined interface (REST)
* Owns its own data (has its own datastore)
* Decentralized data-management (uses event-driven architecture)
* Leverages automation (automated testing and deployment)
* Smart endpoints, dumb pipes (no transformation or ESB middleware)

### What are the benefits of microservices?

* Enables continuous delivery and deployment of large complex applications
* Services are small and easily maintained/testable (maintainability, testability)
* Services are independently deployable (deployability)
* Services are independently scalable (scalability)
* Enables teams to be autonomous
* Allows for easier experimentation and adoption of new technologies (agility)
* Better fault isolation (reliability)
* Faster time-to-market / decreased cost

The implementation of microservices can increase the number of components to manage and create a more complex environment. The application of a microservice layer pattern can help by grouping microservices into a logical bounded context.

Microservice layer pattern:

<https://patterns.arcitura.com/microservice-patterns/design_patterns/microservice_layers>

Microservices can implement a dedicated data source to meet performance requirements

<https://patterns.arcitura.com/microservice-patterns/design_patterns/dedicated_microservice_database>

## Coupling and Cohesion

The two concepts “Coupling” and “Cohesion” have been common when describing the boundaries of microservices.

**Coupling** - The degree to which components have knowledge of other components. Tight-coupling means that changes to resources or contracts on one microservice not only affect itself, but also any/all consumers of its data. In order words to change one thing requires a change in another.

**Cohesion** - The degree to which the elements within a component belong together. Think single responsibility principle and single reason to change. Refers to the limit of dependencies. Cohesion minimizes the number of components in play.

**Temporal Coupling** - Also referred to as time-based coupling occurs when an API expects an immediate response from another before it can continue processing. APIs that perform monolithic orchestration are an example of temporal coupling. Temporal coupling should be avoided in a Microservice Architecture. In cases where state or process must be adhered to, a ‘Process Flow’ API can drive the process as opposed to multiple microservices each representing a specific coupling.

The goal is to achieve low-coupling and high cohesion. It is easy to create a distributed monolith using the microservice architecture pattern if the concepts of coupling and cohesion are not considered. If incorrectly architected, an application based on microservices might end up being a distributed monolith where microservices are dependent upon each other.

Characteristics of a distributed monolith:

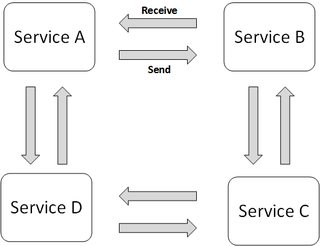
* Changes to one microservice required a changes to others
* Developers work on the codebases of many microservices
* Microservices share the same database or even the code

One way to resolve this issue is to adopt Domain Driven Design which restricts the scope of microservices. The boundaries of the microservice must be correct and adhered to in order to provide for high cohesion and low coupling. If an API needs to wait for another API response then there is temporal coupling. The use of events can be used to prevent the distributed monolith and temporal coupling.

## Orchestration (legacy and transitional) vs Choreography (target)

As the understanding of these patterns are important to the understanding of both the target and transitional reference architectures they are briefly described below. The target API Reference architecture applies the pattern of **Choreography** while the Orchestration pattern is one that exists currently and will continue to exist during the transitional architectures.

### Service Orchestration - Centralization

* Service orchestration represents a single centralized executable business process (the orchestrator) that coordinates the interaction among different services.
* The orchestrator is responsible for invoking and combining the services.
* The relationship between all the participating services is described by a single endpoint (i.e., the composite service). The orchestration includes the management of transactions between individual services. Orchestration employs a centralized approach for service composition

### Choreography – Decentralization

* Service choreography is a global description of the participating services, which is defined by the exchange of **messages**, rules of interaction and agreements between two or more endpoints.
* Choreography employs a decentralized approach for service composition

Microservices architecture favors service choreography over service orchestration, primarily because the architecture topology lacks a centralized middleware component

*notes*

## Event Driven Architecture *(enabling Choreography)*

Event Driven Architecture supports the SOA event-driven messaging pattern in which the consumer establishes itself as a subscriber of the service. The service, in turn, automatically issues notifications of relevant events to this and any of its subscribers.

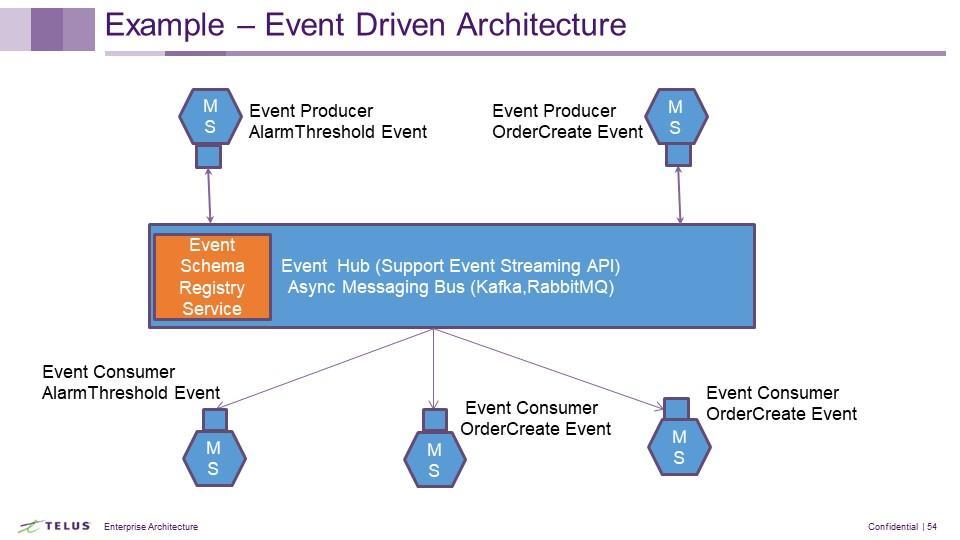
Applying the event-driven messaging pattern:

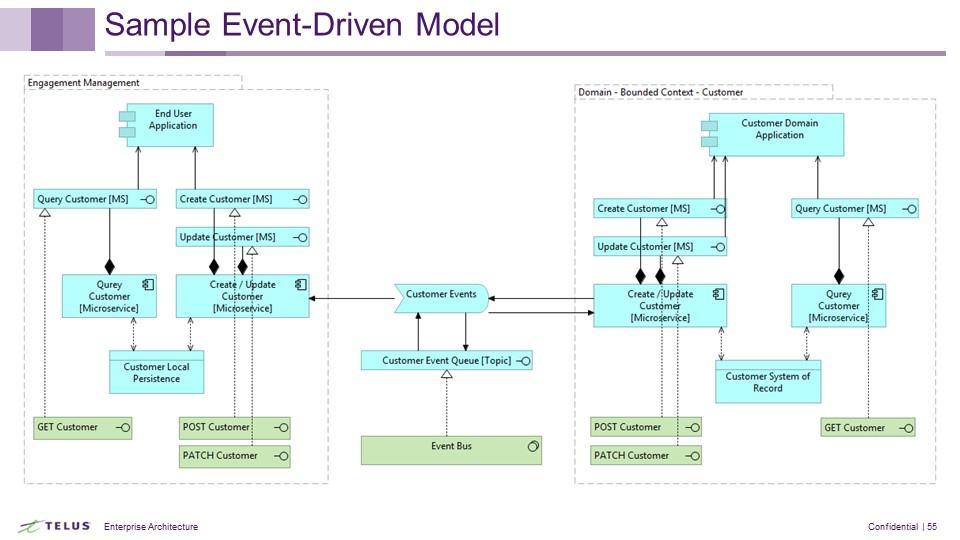
* The client notifies the system that it is a subscriber using typical messaging patterns (e.g pub/sub)
* Systems publish necessary notifications
* Since it is a subscriber, the client can receive notification from the system that is publishing messages
* Implement well-defined events

TM Forum Open APIs support the capability of publishing events to subscribers using a pub/sub paradigm over REST. The current pub/sub paradigm uses a callback mechanism to deliver events to the subscribers however the next version of the specification moves towards support for the publish messaging paradigm. TM Forum Open APIs follow a normalized structure described in the TM Forum Design Guidelines.

Event streaming API will support large scale publication of, and subscription to, events. Event streaming will support multiple event types from microservice to microservice enabled event driven architectures. The Open API event model provides a definition of events, their payloads, and their metadata using the concept of schema defined events and building on the TM Forum application of the polymorphic design pattern. The event hub can be implemented using scalable event frameworks such as Kafka or RabbitMQ.

The Reference architecture does not specify the use of any specific technology being vendor technology agnostic, therefore references to specific technologies are only for example. Specific recommendations and guidelines on technology can be found in the best practices guidelines.





## 

### TM Forum Event and Process Flow API

The Event API provides a standardized client interface to the enterprise event management system for creating, managing and receiving service related events to (indicatively) drive automation workflows, notify other service providers for service outages and SLA violations, trigger Trouble Ticket creation, and enable more complex orchestration scenarios between management systems. The Event Management API can also be used to convey business level Events in support of other processes.

Benefits:

* Standard interface for all events
* Enables event normalization
* Enables event interoperability
* Provides standard language for enterprise integration and communication
* Abstracts the technology of the message bus

The process flow API provides a standard mechanism for handling a longer process that requires a sequential and stateful execution.

## Aggregates

While microservices break a monolithic architecture into small discreet parts that can be managed independently from a business perspective a group of related entities often need to be treated as a whole. From a back end perspective we want to split capabilities into microservices in order to take advantage of cloud native capabilities to improve resiliency, security and reliability. From a front end perspective an application often wants all the data return by a single API. Aggregates provides the ability to group highly cohesive entities into a single, atomic unit. Aggregates work well in a microservice architecture in order to hide the decomposition of capabilities but provide for the front end perspective.

Aggregates can take the form of a microservice which brings together data from a number of other microsevices. Aggregates are used to retrieve data for a front end but should not be used to manage the data of the aggregate entities.

Two common ways of providing an aggregate is through another API or a graph application.

### Microservice vs Graph

Microservice

* Pre-aggregated data
* Changing the aggregation requires a change to the microservice
* Useful when data needs to be pre-aggregated to provide fast access
* Use for stable aggregates

Graph

* Dynamic data aggregation. Read from underlying microservices when requested
* Data is usually cached providing good performance
* Changing the aggregate is done by changing the graph
* Use for dynamic aggregation or aggregates that may change due to a change front end engagement
* Can be used to map underlying microservice data to front end context

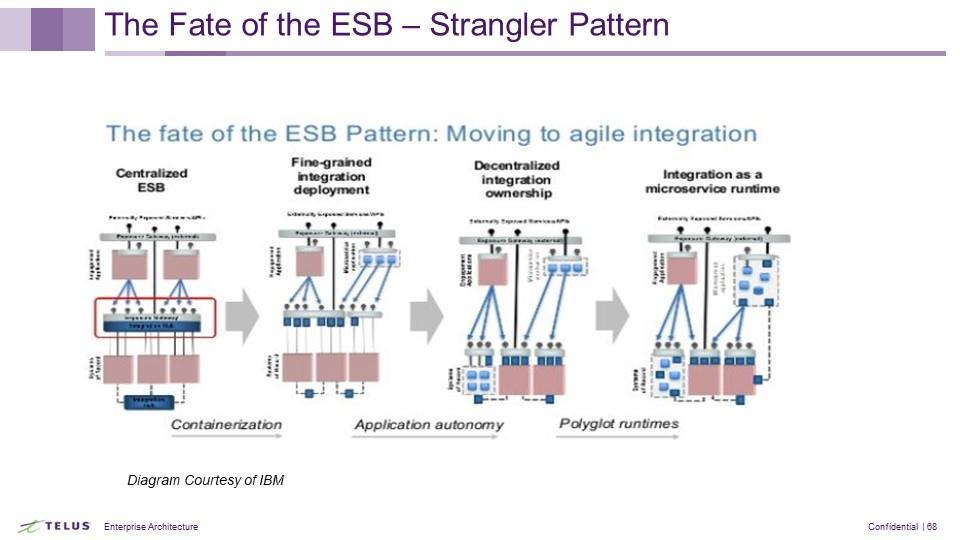
## 

## Eventual Consistency

Refer to Eventual Data Consistency under Data Management and Data Quality section.

## Moving on From Enterprise Service Bus

The target architecture does not include the traditional heavy centralized enterprise service bus. Transitional architectures enable the movement away from heavy weight ESB towards lightweight dumb pipes connected to lightweight orchestration and choreography engines such as Zebee, Kafka, Step Functions, and Open Whisk. The target architecture exposes messaging and event APIs for asynchronous message processing.



# 3. Transitional Architectures

This section discusses the architecture plateaus (phases) which can be targeted on the journey towards the target architecture and provides a viewpoint of each architecture phase as a reference. Transitional architectures focus on breaking down monolithic applications through the application of the principles and patterns of the reference architecture. Each architecture plateau may realize one or more of the principles and patterns. These transitional architectures are provided to assist architects in designing solutions that move their portfolio towards the realization of the target architecture. It is recognized that each portfolio, being in its own unique architectural state, may use the patterns within the reference architecture to realize a different plateau than the models provided. A portfolio’s transitional architecture must still correctly apply the chosen patterns and concepts.

## Principles applied in Transitional Architectures

* Cap Theorem
  + C: consistency is equivalent to having a single up-to-date copy of the data; all clients always have the same view of the data
  + A: high availability of the data; all clients can always read and write
  + P: partition tolerance to network partitions (in distributed system deployments)
* Align microservices to the domain context map
* Create MicroServices to expose data
  + Use the applicable TM Forum Open Data Model
  + Apply the TM Forum Open Model Extension Pattern to include TELUS specific attributes
  + The TMForum standard provides for an interface that is easily consumable by developers and architects
* Segregate Query and Update Functions into different Microservices
* Expose normalized events aligned with TM Forum Event Model on the Event Bus
* Create Master and Slave Microservices and integrate them via Events published and consumed on the Event bus

## Transition Strategies

What strategies can be used to transition from a large, monolithic system to an API Architecture that is microservice-enabled and cloud native?

This section will discuss:

* transitional architectures (plateaus)
* The Strangler Pattern -How to transition the monolith to the target architecture
* On premise (current) towards Enterprise and Cloud

### What is a Transitional Architecture

Transitional architectures reflect a series of steps in order to transition from the current architecture to the desired target. It is important to note that transitional architectures are a means of achieving the target state; they are not themselves the target state. It is therefore critical to have a roadmap that details the steps in the migration from current to target in order to avoid the issue where a transitional architecture is adopted as the target state.

### The Strangler Pattern: Incremental Migration

|  |  |
| --- | --- |
| The Strangler Pattern allows for the incremental migration of a legacy system by gradually replacing specific pieces of functionality with new applications and services. As features from the legacy system are replaced, the new system eventually replaces all of the old system's features, strangling the old system and allowing you to decommission it. The Strangler Pattern is a key pattern to move from an enterprise composed of many complex legacy systems to simpler integration.  The strangler pattern is the main pattern used to break down the monolithic applications and move towards a microservice architecture. Microservices play a role in the transitional architectures. Functionality of the application is analyzed and small parts are identified to be replaced by new services. Creating the business architecture, identifying the domain and bounding the context is the starting point to being able to identify which parts of the application(s) can be transitioned to the new architecture. |  |

#### Benefits of the Strangler Pattern

* **Step-by-step Migration:** Small steps attract less risk, allow roll back, and limit the impact of any problems
* **Limited scope projects:** Taking advantage of decoupling, this approach limits the scope and impact of a project, so it is possible to have multiple projects running in parallel; only limited coordination and synchronisation of projects is required. It therefore fits well with agile development environments
* **Fast time to market:** The pattern can be used to introduce new functionality quickly, without the introduction complexity. Usually new functionality introduced quickly increases complexity and degrades the architecture however done correctly the strangler pattern uses the introduction of new functionality to take a step towards simplicity of the architecture.
* **Increased Flexibility:** By migrating towards common components, flexibility of products and services is improved as a new feature immediately becomes available across the entire portfolio

#### Migration Steps Using the Strangler Patterns

The following outlines the steps to migrate from a monolithic application to a target architecture using the Strangler Pattern. In this example there are three legacy systems that perform the same capability. Each legacy system is integrated with its own system application using a point to point integration pattern. The steps provided will show how to migrate from the current state to the target state of one strategy system for the capability while continuing to add new functionality.

|  |  |
| --- | --- |
| **Step 1 - Insert An API**  In the first step an API, such as a TMForum Open API is inserted between the Legacy SM1, SM2 and SM3 and the upstream applications System 1, System 2 and System 3. This moves the upstream applications to a standard interface for retrieving information.  If we consider these systems to be Product Ordering, the API would be the TMForum Product Ordering Open API. Each upstream system would now consume this API. |  |
| **Step 2 - The Virtual Component**  The second step in the migration is to make all of the three legacy systems (Legacy SM1, SM2 and SM3) act as a virtual management system. In the case of Product Ordering all three would act as a virtual Product Ordering system to the upstream applications. All three applications; System 1, 2 and 3, in our example call the same API. Behind the API there is a routing function that routes the calls to the appropriate legacy system. The API uses the polymorphic pattern. This design pattern of the API interface allows the API to carry the payload required by each system. The API can also contain any shared logic, common validation, data massaging, etc. |  |
| **Step 3 - Strategic Component (Target System)**  In the third step a strategic component is identified. The strategic component represents the target system. This does not necessarily mean a new system. One of the current legacy systems could be identified as the target strategic system.  In the example a new system Strategic SM is added. |  |
| **Step 4 - Migration To The Strategic Component (Target Component)**  Migration to the new strategic (target) system begins service by service (function by function). The first functionality may be new and not supported by the legacy. The routing table takes care of routing the requests to the new strategic system. |  |
| **Step 5 - Migration To The Strategic Component (Target Component)**  As functionality moves to the new strategic system, legacy systems can be retired. The timeline for migration depends on many factors including the timeline for retirement of legacy systems. However the roadmap for migration is created the changes should be invisible outside the boundaries of the api. Upstream systems should not be impacted when using the Strangler Pattern. |  |
| **Step 6 - Migration Complete**  Eventually all the legacy systems can be retired leaving the single strategic system. |  |

#### 

## Coexistence Strategies

Transition to the target architecture will take one or more steps therefore coexistence strategies are required. The strangler pattern discussed previously is, during transition, a coexistence strategy. While being used to transition to the target architecture, parts of the application still remain in the legacy state while other functionality of the application is created in the target state.

The exposure strategy of Façade APIs for the bounded-context, as demonstrated in the Strangler Pattern, helps abstract the internal applications of the platform from its consumers enabling the internal transformation of the platform without impacting consumers. Consumers communicate to the platform through the well-defined interface of the Open APIs. These interfaces are decoupled from the implementation and therefore do not change even though the implementation may change from the legacy monolith to the target architecture of a microservice. The initial steps of the Strangler Pattern diagram above shows the implementation of a Façade APIs.

#### Other references for Coexistence Strategies

##### Legacy Wrapper

<https://patterns.arcitura.com/soa-patterns/design_patterns/legacy_wrapper>

##### Multi-Channel Endpoint

<https://patterns.arcitura.com/soa-patterns/design_patterns/multi_channel_endpoint>

## The Role of an Enterprise Service Bus in the transitional architectures

The ESB, being part of the legacy architecture, continues to play a role in the transitional architecture until complete transition to microservices. It provides a mechanism for creating a normalized response to upstream applications, and provides orchestration of many downstream services and applications across domains. Some legacy applications or specific use cases may best be served by this centralized approach, however the increasing need to distribute service capabilities, moving capabilities to the cloud and edge of networks, does not. The ESB therefore plays a transitional support role towards the target architecture and is eliminated in the target.