ONAP Microservices - Approaches

Manoj Nair
NetCracker
This presentation’s focus
• Improvements to Model Driven Approach proposed in Alex’s presentation, with reference to some best practices
• How Modularity can be achieved (Reference to Alex’s proposal earlier) : Best Practices from TMF
• : Approaches and Views
• Cloud Native Micro Services
References : https://wiki.onap.org/display/DW/Microservices+Industry+Perspectives
What is proposed here?

• Three approaches to refine Microservice Architecture
• Microservice Architecture: Deals with how microservices are defined, realized and how they interact.

• **Approach 1: Model Driven Microservice – Approach based on DDD**
  - Other Open source implementations – ODL, OSM etc, Model driven Microservices by Alex, Reference to OSM implementation.

• **Approach 2: Modular Microservices - Leverage TMForum IG1118 Management and Control Continuum and Future Mode of Operation concepts**
  - Based on proposal by Alex and Ramesh on DMS, but enhancing it further with reference to a standard architecture.
  - Primarily focused on structuring microservices and components in a hierarchical “Lego blocks” like units which can be mixed and matched to create management and control functions.

• **Approach 3: Cloud Native Micro Service Design (Followed by MSB)**
  - Based on the Service Mesh technique used in the cloud native world.
  - Mainly an implementation approach than a consistent microservice design methodology.

Above approaches are complementary to each other and not exclusive.
Approach 1: Model Driven Microservices
Reference: Model Driven Micro Services (Alex)
Domain Driven Design Concept

- Top down approach of designing Microservices
- Domain-driven design (DDD) advocates modeling based on the domain of business
- In the ONAP context Domain may be - Hybrid Telecom Service Orchestration and Management
- In the context of building applications, DDD talks about problems as domains
- It describes independent problem areas as Bounded Contexts (each Bounded Context correlates to a microservice)
- Bounded contexts own its own domain data, logic and behavior
- In DDD to define the domain model for each Bounded Context,
  - identify and define the entities (and other patterns if required)
  - create a boundary around things that need cohesion
  - build and refine domain model contained within the boundary that defines the context
  - Each bounded context can be modelled as a microservice
Domain Entity Pattern

• Entities represent domain objects and are primarily defined by their identity and not only by the attributes that comprise them.

• An entity’s identity can cross multiple microservices or Bounded Contexts.
  - For example, Policy in one microservice might be TCA Configuration in another microservice or Constraint in yet another microservice.

• The same entity can be modeled across multiple Bounded Contexts or microservices as different identities.

• However, that does not imply that the same entity, with the same attributes and logic would be implemented in multiple Bounded Contexts.

• Instead, entities in each Bounded Context limit their attributes and behaviors to those required in that Bounded Context’s domain.

Reference: link
An example

Bounded Contexts

- DCAE
- Closed loop policy
- OOF
- Constraint

Each bounded context owns the management of identities

- PAP
- Configuration Policy
- PDP/PEP
- Operational Policy
- Configuration Policy

All refers to same entity with different identities, different attributes and behavior
ONAP’s current MS architecture is defined per component. So the domain as per DDD has to be looked at a per component level:

- ONAP seems to limit domain scope to each component OR sometimes seem to refer to use cases in the context of domains.
- Microservices are bound to components than end to end business context (not loosely defined based on domain entities but bound within each component) or use cases.
- To realize business use case it has to be disintegrated at component level and then MS level.
- Multiple components and associated microservices need to be augmented to support end to end capability.
- In ONAP each component bounds the entities, domain data and associated identities. This creates a dependency and violates the independent development practices put forth by DDD through bounded context.
- Kubernetes seems to map microservice to Service and not Pods, but currently each component in ONAP is mapped to Kubernetes Service with associated Pods representing docker containers fulfilling the micro functionality. Reference: [link](#), [link](#)
How DDD enables Model Driven Design

- Encouraged to create domain entities and bounded contexts (Quite similar to what modelling team is doing with CIM, but might require further identification of bounded contexts)
- Each bounded context has own domain model and ubiquitous language relevant in own boundary
- A context map is formed representing interaction between bounded contexts
- Microservice boundaries are defined based on bounded context
An Abstract View of Realizing DDD

Bounded context

ONAP Microservice

Catalog

Inventory

Workflow

Policy

Distributed Data Store

Constraint

Configuration Policy

Closed Loop Policy

Action invocation/API End Point Registration

Pub/Sub

API Endpoint Registration

Common Information model
Example – Model driven approach by OSM

- Models are centrally managed
- Yang models are compiled to generate base classes used by all the components.
- IM-NBI uses pyang tool set to generate python code, json schema and json message templates to be used by modules in OSM
- Centralized management of data models helps in having a unified view across OSM and extensions are managed centrally
Micro services register with MD-SAL platform – about capabilities, interfaces, entities etc. Each MS is modelled using YANG and compiled to create skeleton code.

OpenDaylight MD-SAL – Framework stores Config and Operational data and enables interaction between MS.

MD-SAL takes care of data storage, cluster data consistency, sharding, interaction between micro services, set of tools for project skeleton. Any change in distributed config/operational tree is notified to subscribed MS.

Reference: Opendaylight MD-SAL Architecture
Model Driven Microservices – One possible implementation view

ONAP Microservice

ONAP Microservice

API

Generated API

Signature

API Invocation (Actions)

Configuration read/sync

Subscription for Data Change, Notification, Configuration and Operational State update

ONAP Primary Instance

Distributed Configuration Store (OOM – Consul ?)

DMAAP/Real Time Data Event Streaming & Processing

External Domain Orchestrator/Controller

API Routing

External System Registration

MS Configuration push to Distributed store

Registration for topic + State change notified to subscribed MS

State update resulting in event
Key Concepts 1/2

- Microservice deployment unit consisting of the core MS software along with metafile which contains configuration and deployment descriptor (Can be modelled using SDC ?)
- MUSIC for distributed, highly available storage for domain entities
- Distributed Configuration Store for maintaining runtime configuration data of MS
- Microservice Bus which act like a MS Endpoint Registry and API call routing GW
- DMaaP or Real time data event streaming bus (based on RESP) which routes messages and file transfers across distributed MSs
- OOM which reads the Metafile of MS and carries out deployment, registration.
1. Micro services are packaged as deployment units – Packaging here is just logical in nature. Physically this can be an ONAP micro service component with a well-known (XML, YAML etc) configuration, deployment descriptors which acts like a metadata for MS. In practical sense the Metadata may be split to two files – deployment descriptor and configuration descriptor. Deployment descriptor is used as input for deployment by Kubernetes or Docker Engine. Configuration Descriptor captures following details
   - MS configuration like DB connectivity details
   - MSB registration including the capabilities
   - DMaaP registration including the topics published and subscribed
   - CHAP related configuration
   - Domain data bound by MS
   - Swagger for generating APIs for MS (which can be consumed by MSB)

2. During the microservice deployment (using deployment descriptor), metadata is read from the microservice predefined directory structure and pushed to the configuration data store. Rest of the configuration changes can be applied on the configuration store

3. OOM deploys MS and carries out registration of MS on MSB based on the registration data read from configuration store. Similar registration mechanism can be carried out for DMaaP as well.

4. MS carries out self-configuration or OOM initiates configuration of MS by reading the configuration data store

5. During the lifecycle of the MS the state is updated on the CHAP distributed DB which triggers notification on DMaaP

6. Notification forwarded to consumer by DMaaS

7. Consumer act upon state change: This is optional – indicating the event driven microservice behavior

8. Consumer potentially invokes action on Provider through an API call: This is optional – indicating the event driven microservice behavior
Approach 1: Observations

• The concepts described in this approach focus on the boundary, structure and interaction patterns of microservices.

• Microservices can be model driven following the domain driven design principle with bounded context mapped to a microservice. Each bounded context maintaining own domain model.

• The reusability of the microservices and aggregation of microservices to form a system or a solution with a well defined grouping pattern is not covered well.

• Require effort in refactoring the existing microservices to define context map, domain models etc.

• Will enforce top down modelling of microservices with alignment to domain rather than independent projects or components.
Approach 2: Modular Microservices Design
- **Consumer Interface**: Interface used by component to fulfill needs for tasks to be carried out related to its purpose. The tasks will include the evaluation of information and the providing of results.

- **Provider Interface**: Interface through which a component offers a capability to carry out specific tasks related to its purpose.

- **Application Function**: Exposed through Application API and provides access to all the related to the primary business purpose of the component.

- **Operations Function**: Exposed through operations API and implements the support for managing the lifecycle process for the component.

- **Security Function**: Exposed through security API and provides security features for the component.

- **Environment Function**: Accessed through the execution environment APIs and used by the component to access the infrastructure resources that it needs to support its functionality or operations.
Management and Control Continuum

• A part of FMO Concept which focus on operational aspects of services in the target architecture built on NFV and SDN.
• Deals with component that have management-control functions as their primary functions.
• In traditional systems functions like EMS/NMS, Control Plane etc are considered as MCC functions.
• In the MCC architecture MCC components can be assembled to form MCC Systems. An assembly of MCC components can be given a certain name/branding.
• A collection of functional components can be packaged together and delivered as a solution to a particular problem.
• Such an assembly in itself can be considered as a component.
• Each component in an assembly is contributing its application function to the capabilities of the overall branded component
• The same MCC component can be offered as part of different branded components.

Reference: TMForum IG1118
FMO Architecture: Composition of Components

Reference: TMForum IG1118
FMO Component: Positioning of a Component in a hierarchy of modular constructs

- A component-system pattern is fractal, split a component and you get more components, combine components and you get a component.
- MicroServices can be perceived as a component fulfilling a cohesive function.
- A Branded Element constitutes of multiple features which in turn are realized through set of functions – all levels can be a composite component.

Reference: TMForum IG1118
FMO Component : Key characteristics

• **FMO component** is a good way of visualizing the Microservices with well-defined boundaries expressed through – Operations, Application, Security and Environment Functions.

• FMO Component supports a provider consumer pattern and interfaces for offering or consuming capabilities that can be expressed through APIs.

• FMO Component may have multiple contexts – information context (e.g.: software, image, descriptors), delivery context (packaging, modules), deployment context (deployable unit – as an assembly of functional components in a deployment unit), catalogue context (represented in a catalogue as a functional component along with delivery and deployment context), runtime context (component artifacts turned into runtime entities consuming resources and exposing APIs).

• For ONAP, MCC Component can be interpreted as the basic unit of functionality that can be represented in a microservice and can be assembled to form a system or a branded component (for example DCAE).

Reference: TMForum IG1118
Relevance in ONAP – Assembly of Branded FMO/MCC Component

ONAP Branded Component (e.g. OOM, MSB, AAF)

ONAP Branded Component (e.g. External API)

ONAP Branded Component (e.g. A&AI)

ONAP Branded Component (SDNC)

ONAP Solution as an assembly of Branded Components
ONAP as an assembly of Solution subsets

ONAP with OSS/BSS MCC Components (E2E Service Orchestration)

One layer’s capability is infrastructure to the next layer

Layer 1

Layer n

ONAP as Domain Orchestrator
Branded Component

ONAP Branded Component

ONAP as Domain Orchestrator
Branded Component
ONAP Modularity and Standard Alignment through Proxy Component

- A branded ONAP component (such as SO or DCAE) can be represented through the interfaces consolidated/exposed by a single component which acts like a proxy.
- This proxy can be realized using ONAP MSB or can be a dedicated microservice within ONAP Branded Component (e.g. side car exposing the component to the service mesh)
- This approach can be used to enable standard alignment. For example if two SDO interfaces need to be supported based on different deployment models, two proxy components can be created with different component interfaces.
- Modularity is ensured by the FMO component structure and the APIs exposed by micro/macro levels of components which are consolidated through proxy component
Suggested changes in ONAP

• Refactoring of APIs across components/Microservices in the Security, Operation, Application and Environment categories (Anatoly from NOKIA: This is not as easy as we think, also there may be some ambiguity which need to be cleared – Operation API for one MS may be Application for another, there is no clear guideline for classifying the API to different categories)

• Have basic minimum set of entities to be supported in microservices to enable manageability and control.

• A means for the components/Microservices to express capabilities in a catalogue to enable appropriate assembly (MSB capability limited to end point registration, not capability registration)

• Governance of interaction between components through policy (Already available through Configuration Policy ? – Policy Driven Orchestration ? )

• To have capabilities built in to components/microservices to achieve lifecycle compatibility (API/Interfaces change and upgrade support) with relevant mediation functions or adaptors

• A mechanism to express different context of Components/Microservices – Delivery context, Deployment Context, Information Context, Catalogue context, Runtime Context etc.
Modularity: How Microservice Composition can be Represented? Learnings from TMForum TAM

- Define Microservice Levels (Level 1, Level 2 etc) that compose domain functions
- Define well defined APIs for each level of Microservice- roughly based on IG1118 categorization.

- What problem this solves:
  - Reduce ambiguity in defining microservice boundary, ensure consistent view for MS.
  - Well defined APIs at each level makes it modular and ensures core properties of MS – Easily replaceable, Can be independently developed and upgraded
Approach 3: Cloud Native Micro Services
Cloud native microservices

- Microservices designed with cloud native principles in mind
- 12 factor principles – [link](#) (need not be cloud native, but general MS principles)
- CNCF currently is tool focused that work with COEs and dockerized containers. Heavy Kubernetes focused
- ONAP already uses some of these tools sets (Kubernetes in OOM, Tracing tools in Logging)
- What is missing? A framework which integrates all these toolsets that can be readily used for implementing microservices (i.e consistently across microservices)
- CNCF so far has not given MS design guidelines, structure or modelling for MS other than what COE provides.
- Interfaces like gRPC/GPB seem to be quite efficient compared to REST APIs. Is there any need to migrate from REST API to gRPC?
How Cloud Native is ONAP?

- Not all Micro services in ONAP are designed with an objective to be cloud-native. Some important characteristics that should be considered are:
  - Leverage the cloud characteristics like scaling, healing and migration (handling lifecycle management operations) (Some components support this – e.g. DCAE)
  - Version and Update/Upgrade handling without impacting the functionality (?)
  - Portable across cloud infrastructure (No dependency on Cloud Infrastructure – Intel vs AMD, AWS vs OS vs GC vs Azure vs VMWare vs COE)
  - Inbuilt state management, consistency checking mechanisms (Project specific, How each project mitigates CAP- Consistency, Availability, Partition Tolerance)
  - Inbuilt health check mechanisms (Supported by OOM ?)
  - In built fault tolerance mechanisms (Supported by OOM ?)
  - Fault isolation capability (OOM/MSB?)
  - Secure interactions – internal/external (aaf?)
  - Tracing of flows across microservices (Planned to be supported by Logging?)

- Not all microservices endpoints plugged into MSB
- Each project use own model of state management and consistency mechanism across distributed instances of MS
- Scaling/Healing and Migration of MS are not consistent and in the early stages.
- No consistent deployment mechanism - Mix of blueprint, heat templates, helm charts, dockerfiles for microservice onboarding
- No consistent configuration formats – Consul, YAML templates, Helm, XML, JSON, HOT
- Version Portability (Amsterdam to Beijing) ? Lossless Software Update ?
CNCF Toolsets that can be leveraged in ONAP – Reference CNCF, IETF

- Service Mesh: Istio, Linkerd, Envoy
- gNMI/gRPC/ProtoBuf as enhancement for VES Collector
- Prometheus for MS/VNF telemetry collection, monitoring
- TUF for Secure SW update
- Notary, SPIFFE, SPIRE
- Fluentd, Jaeger, OpenTracing
- Rook for Cloud native distributed storage, which can be leveraged in MUSIC

- Open Policy Agent
- gNMI/gRPC, QUIC for NE/VNF Management
<table>
<thead>
<tr>
<th>Project</th>
<th>Relevant application in ONAP</th>
<th>Maturity level</th>
<th>Backing companies</th>
<th>Recommended ?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kubernetes (COE)</td>
<td>OOM, DCAE</td>
<td>High</td>
<td>Google + Many others</td>
<td>Yes. But troubleshooting, monitoring, deployment seem to be complex.</td>
</tr>
<tr>
<td>Prometheus (Monitoring)</td>
<td>DCAE, OOM</td>
<td>Medium</td>
<td>Robust Perceptions, Container Solutions, Independent Contractors</td>
<td>Can be used for container monitoring at least for time being. Roadmap is not clear.</td>
</tr>
<tr>
<td>Fluentd (Log data collector), Jaeger (Distributed trace analysis), OpenTracing (Distributed trace collector)</td>
<td>Logging</td>
<td>Medium (currently at 1.0 Release)</td>
<td>Jaeger from Uber, Opentracing – Lightstep. Github contributors – Fluentd(135), Jaeger(45)</td>
<td>Distributed tracing using Jaeger or OpenTracing might be useful for ONAP. (Requires instrumentation in code)</td>
</tr>
<tr>
<td>Envoy (edge proxy, Service Mesh Data plane)</td>
<td>MSB, ONAP Projects, OOM, AAF</td>
<td>High</td>
<td>Strong contributor base (161), Originally built by Lyft</td>
<td>MS proxy might be relevant while adapting service mesh. But need to debate if service mesh is good enough for ONAP. Istio Service Mesh CP works well with envoy.</td>
</tr>
<tr>
<td>Linkerd (Edge proxy, Service Mesh Data Plane)</td>
<td>MSB, ONAP Projects, OOM, AAF</td>
<td>High</td>
<td>Healthy contribution base (70), Buoyant</td>
<td>As per this comparison Linkerd seems to be feature rich, but consumes more CPU and memory compared to Envoy. Supports HTTP/1.1, Thrift, ThriftMux, HTTP/2 (experimental) and gRPC (experimental). MSB team to comment suitability of Linkerd.</td>
</tr>
<tr>
<td>Istio – Service Mesh Control plane</td>
<td>MSB, OOM ?</td>
<td>High</td>
<td>Google, IBM, Lyft.</td>
<td>Preferred if there is a plan to adapt Service Mesh in ONAP. How this will be integrated? As a separate project? As part of MSB/OOM?</td>
</tr>
<tr>
<td>Project</td>
<td>Relevant application in ONAP</td>
<td>Maturity level</td>
<td>Backing companies</td>
<td>Recommended ?</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------------------------------------------------------------------------------</td>
<td>--------------------------------</td>
<td>------------------------------------------------</td>
<td>-------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Notary (secure content sharing), TUF (secure software update spec)</td>
<td>AAF (Not sure), VNF SDK, OOM (software update – to be confirmed), SDC Package manager (to be confirmed)</td>
<td>Low (Not yet in a major release)</td>
<td>Healthy contributor base (60+)</td>
<td>AAF team or Security subcommittee to comment. Might be relevant for software upgrade, image upload, Marketplace, VNF SDK (all external facing)</td>
</tr>
<tr>
<td>Vittes (Clustering of MySQL, Sharding)</td>
<td>Many projects – that use MySQL or MariaDB</td>
<td>Medium</td>
<td>Healthy contributor base (100+)</td>
<td>Project teams to comment. May be relevant for distributed deployment of ONAP components, Scalability enhancements etc.</td>
</tr>
<tr>
<td>Rook (Cloud native distributed persistent store)</td>
<td>Music, Other projects that require distributed persistent store</td>
<td>Low (Not yet in major Release)</td>
<td>No known companies. Good contributor base (70)</td>
<td>Not recommended. Low maturity and no known deployment.</td>
</tr>
<tr>
<td>SPIFFE (Spec)/SPIRE (Implementation) – Service Identity Mgmnt</td>
<td>MSB, AAF (Service Identity Management)</td>
<td>Low</td>
<td>Scytale, Istio</td>
<td>No. Similar capabilities in MSB and AAF. But low maturity level. Istio-Auth seems to incorporate this. AAF and MSB to comment.</td>
</tr>
<tr>
<td>Open Policy Agent</td>
<td>Policy, MSB</td>
<td>Low</td>
<td>No known companies</td>
<td>No. This is primarily meant for controlling the MS behavior through policy with a daemon deployed alongside the service. Not recommended immediately due to maturity issues. Based on maturity can be considered to position this as a PDP per component (Policy team to confirm)</td>
</tr>
<tr>
<td>NATS (NAT Server) - Messaging, Publish/Subscribe, Request Reply, Queuing</td>
<td>MSB</td>
<td>Medium</td>
<td>Healthy contributor base – used by Ericsson, HTC, Siemens, VMWare</td>
<td>Require evaluation. Similar capabilities like MSB. Supports different types of communication models. Uses a proprietary messaging format (gnatd) which is claimed to be efficient. May be useful if we want to unify communication models under single project (but will require lot of refactoring)</td>
</tr>
<tr>
<td>Core DNS (DNS functionality, Service discovery)</td>
<td>MSB, DNS VM</td>
<td>Medium</td>
<td>Healthy contributor base (80+). Used by Sound Cloud, MIT</td>
<td>Require evaluation. Already part of K8S v 1.9 onwards. Need to see if the scope is limited to K8S cluster and containers.</td>
</tr>
</tbody>
</table>
# gRPC/gNMI/gNOI Views

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>gNOI</th>
<th>gNMI</th>
<th>gRPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is the focus</td>
<td>gRPC based Operational interfaces to network devices</td>
<td>gRPC based protocol for modification and retrieval of configuration from target device</td>
<td>Generic RPC mechanism</td>
</tr>
<tr>
<td>What is available?</td>
<td>Microservices which can be reused for enabling the interface</td>
<td>Client library implementing network management interface</td>
<td>Many implementations in different languages</td>
</tr>
<tr>
<td>Who is driving?</td>
<td>Open Config</td>
<td>Open Config</td>
<td>Driven by CNCF</td>
</tr>
<tr>
<td>Contributors ?</td>
<td>Limited</td>
<td>Limited</td>
<td>Many</td>
</tr>
<tr>
<td>Where is it relevant in ONAP</td>
<td>SDNC, App-C, VF-C, DCAE (all SBI)</td>
<td>SDNC, DCAE, App-C, VF-C (all SBI)</td>
<td>Between MS, ONAP and external systems</td>
</tr>
<tr>
<td>Dependency</td>
<td>gRPC, protobuf</td>
<td>gRPC, protobuf</td>
<td>gRPC Stub/Service implementation, protobuf. HTTP/2</td>
</tr>
<tr>
<td>Why it is relevant?</td>
<td>gRPC based, efficient communication, ready to use operational protobuf specs</td>
<td>gRPC based, efficient communication, ready to use operational protobuf specs</td>
<td>Bidirectional, Asynchronous, Can be used for notifications or RPCs, low latency, can support many communication patterns, Can also support REST HTTP1.1 (proxy) . Very low dev overhead.</td>
</tr>
<tr>
<td>Recommended for ONAP</td>
<td>Need to check how many VNF vendors support this. <strong>May not be immediately.</strong> SDOs do not seem to endorse this yet</td>
<td>Need to check how many VNF vendors support this. <strong>May not be immediately.</strong> SDOs do not seem to endorse this. yet</td>
<td>May be some wrapper rather than exposing gRPC directly to MS? (Service Mesh ?)</td>
</tr>
</tbody>
</table>

All options above may require Kubernetes
<table>
<thead>
<tr>
<th>Characteristic</th>
<th>gRPC</th>
<th>REST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>Impressive</td>
<td>Relatively slow</td>
</tr>
<tr>
<td>Development overhead</td>
<td>Low (Stubs can be auto generated)</td>
<td>High</td>
</tr>
<tr>
<td>Limitation</td>
<td>Version management, Compatibility, Requirement for retooling, Stub/Service Management</td>
<td>Processing overhead</td>
</tr>
<tr>
<td>Compelling capabilities</td>
<td>Low development overhead, performance, communication patterns</td>
<td>Wide adoption, tool sets, Developer familiarity</td>
</tr>
<tr>
<td>Where it best fits</td>
<td>Internal ONAP MS communication</td>
<td>ONAP External communication</td>
</tr>
</tbody>
</table>
OMSA-ONAP Microservice Architecture

OMSA is the vision of ONAP Microservice Architecture to support carrier-grade requirements of ONAP Microservices, which includes service registration/discovery, service communication, API gateway, service orchestration, service governance and service monitoring.

Note: this diagram is a functional view of OMSA, which is not mapped to specific projects.
OMSA Observations

• OMSA is primarily an implementation approach than dealing directly with the Microservices design principles – Modularity, Model Driven, Independent development, consistency concerns.

• Limited Scope – i.e covering the Micro Service discovery, Interaction. How about nature and boundary of microservices?

• Summary:
  - Good for short term benefits
  - Scope of MS should cover more than just inter-MS interaction and discovery
Summary of views

• All approaches bring-in significant practical value

• Among the three approaches, Approach 2 seems to be more favorable considering the ease of adoption and the modularity that can be achieved
  - Approach 2 might be favorable from a component level standard API and model alignment point of view as flexible proxy components can be supported
  - Approach 2 can also leverage some of the concept of bounded context pattern described in approach 1
  - Approach 1 might require a major revamp across projects - the way Model states are maintained, propagated and acted upon. So this approach may be considered for long term.

• Approach 3 is more of an implementation approach with focus on discovery and interaction of MS. MS still need to be structured well, consistent and follow cloud native principles to leverage benefit of this approach.

• Approach 3 seem to give some immediate benefits and may be considered for short term, while other two approaches may be considered for future releases.
Recommendations for Casablanca


- Model Driven Micro Service : (Can be for long term- based on community view)
  - Central model repository referenced by all components
  - Common library of model base classes used by components
  - Anything Else ?

- Modularity of Microservice (Can be for long term- based on community view)
  - API Refactoring – Consistency in representation of APIs across MS and components
  - Policy tailoring all interaction between microservices/components.

- Communication between Microservice
  - Service Mesh or gRPC or RESP or DMaaP or REST ?

- Testability of ONAP components based on Cloud Native Principles (enhance S3P)
- Define guidelines for creating cloud native microservices
- Microservices Meta Modelling through SDC ? (to ensure consistency)
Recommendations for Future Releases

• Best practices incorporated from Approach 1 and/or Approach 2

• How Microservices Architecture support following
  - Different deployment models – Brown field, Green field
  - Different Integration models – point to point , geo distributed
  - Different testing models – unit testing, integration testing, CI/CD

• How following requirements are supported
  - In service software update
  - ONAP release upgrade
  - Model/Policy/Workflow tuning
  - Troubleshooting – Remote, Local, ONAP specific OAM KPIs