ONAP Modularization

AT&T – Vimal Begwani, John Jensen, John Ng

Huawei - Chaker Al-Hakim, Margaret Chiosi, Seshu Kumar

Goal and Agenda

Goal: Evolve ONAP to a more modular, agile architecture:

- ❖ Breaking ONAP components into smaller reusable modules
- ❖ Enabling technology swap-out for modules
- ❖ Reducing software footprint
- ❖ Allowing integration of non-ONAP components

Agenda

- Key ONAP challenges and critical gaps
  - Issues identified by the broader ONAP user community
- Definitions of key terms
- Architecture principles and approaches as a guide to address challenges
- Refactor ONAP by leveraging common services to the fullest extent possible
- Approach: Focus on one major ONAP component at a time
- Examples: Service Orchestrator and Controller

Problem Statement

- ONAP is too complex, too big and hard to make changes.
- ONAP Components are monolithic (SDN-C, SO) and large, not sharing common utilities
- Service providers might have a specific module already implemented and would like to integrate that module into ONAP
  - External controllers (e.g. VNFM, SDN Controller), external orchestrators, collectors, analytic microservices
- Service providers would like to deploy ONAP incrementally, whereas today ONAP supports all-or-nothing approach
  - Core components of ONAP such as SDC, SO, and A&AI must be deployed
  - Other components can be added on as needed basis, depending on the scope of use
- Should ONAP modules migrate to cloud-native microservices?

Can incorporate additional issues and/or more details if available

Definitions of Key Terms

- **Module:** Implements a business capability accessed through a defined set of APIs
  - E.g. A DCAE Data Collector microservice, A&AI data repository
- **Component:** A collection of modules that are related in some form
  - E.g. SO, Controllers, A&AI, etc
- **ONAP:** A collection of ONAP Components
- **Microservice:** Small, single-capability focused, standalone services
  - E.g. IP address assignment, Tosca parser
- **Cloud-Native**: Container-packaged, dynamically managed, microservices-oriented applications
  - E.g. Containerized microservices managed by Kubernetes
- **Service Mesh**: Connective tissue between microservices
  - E.g. traffic control, resiliency, security, observability
  - Control plane (Istio, linkerd) and Data plane (Envoy, linkerd)
  - Note: This is different from service chaining

Approach: One component at a time

Evolutionary To Maintain Backwards Compatibility (Rather Than Greenfield Approach)

Approach

1. Focus on solving component-specific problems
2. Adhere to principle of Refactoring
3. Validate new technologies on selected areas before broad use
4. Progressively build a platform of reusable technologies
   - Establish project to collect Common Services over time
5. Focused partnership with selected PTLs to validate and refine our approach
6. Learnings from initial implementation will benefit subsequent module conversions
7. Maintain backwards compatibility

Avoids

1. Massive undertaking of decomposing all of ONAP into functional elements in one go
2. Unnecessary disruption to ONAP User Community and Planned Release Delivery

ONAP Architecture Principles Applied

- Lifecycle Support
- Standardization
- Pluggable Modules
- Integration Friendly
- Backwards Compatibility
- Microservices
- Shared Services
- CI/CD Support
- Integration/Std APIs
- Cloud Env Support
- Scalability
- Availability/Resilience

High level view of functional approach and SO functional decomposition plan

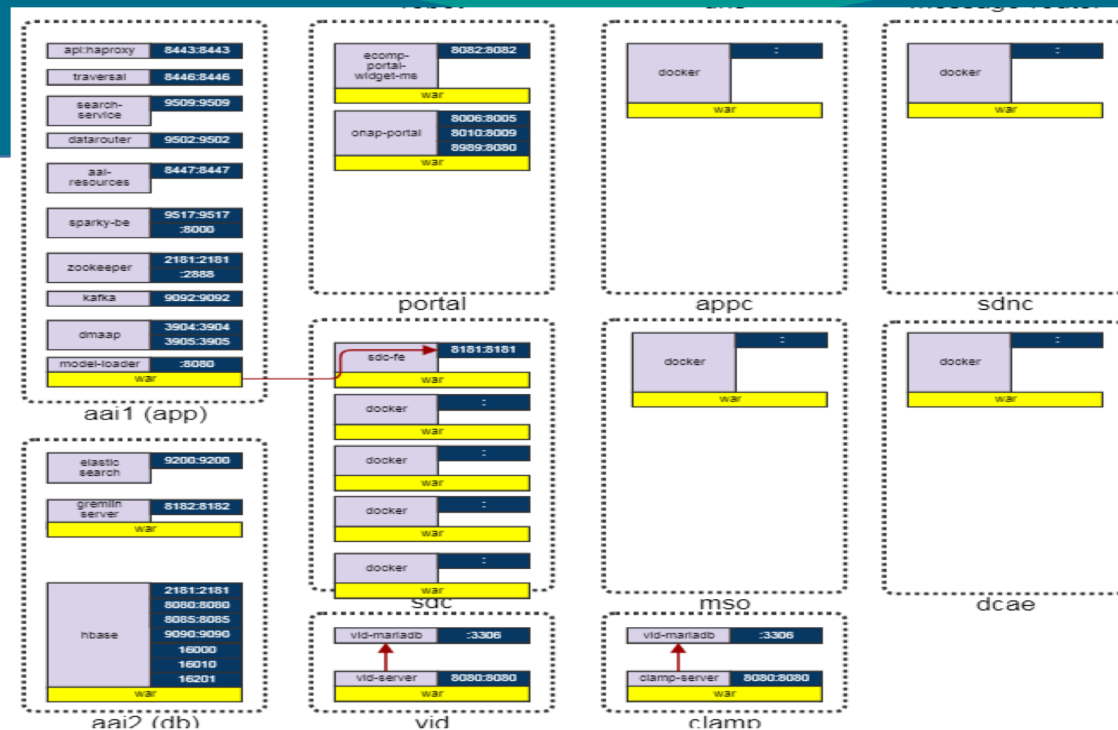
Modularity in ONAP

High level view of functional approach and SO functional decomposition plan

Impacts and Issues in Current ONAP

ONAP modules are currently as

- Individual docker containers that are mostly bulky
- Confined to a set of functionalities that are not allowed to develop in parallel.
- Not re-usable for a part of the functionality if required
- Modules have its own confined boundary space with no transparency
- Duplications across modules like Tosca Parsers / Dynamic Catalog …
- In short Monolithic non-replaceable and non re-usable components

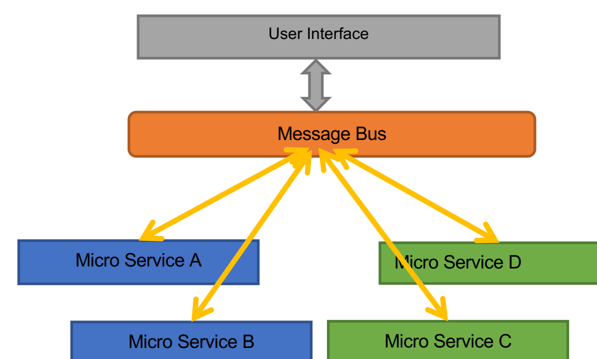


ONAP Architecture from Deployment Perspective

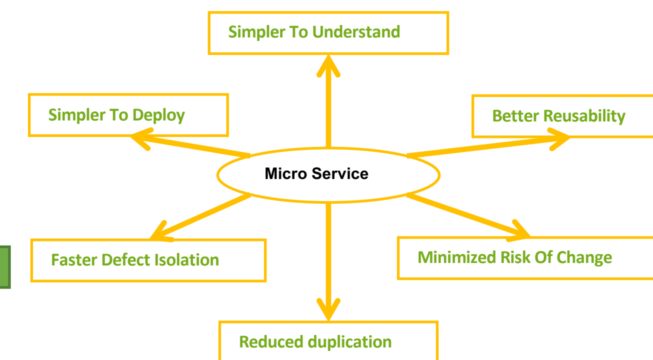
Proposed Solution:

Microservice Based Architecture :

- ✓The idea is to get in the separation of concerns through individual developed modules that could interact with each other through a set of APIs and expose the desired functionalities
- ✓These functionalities can thus be re-used across the ONAP as-is-needed basis
- ✓Current ONAP components would be needed to be segregated to smaller functional deployable units with their interaction driven through APIs



Typical Microservice Architecture



Advantages of Microservice Architecture

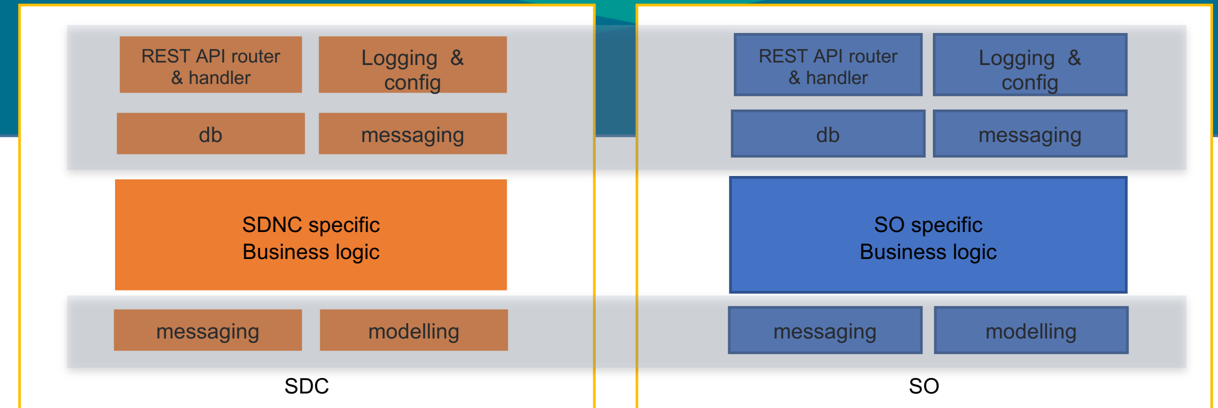
Extensibility

Problem statement

In ONAP, every components implements common non-functional aspects such as REST API handling, logging, configuration, db handling, messaging on its own and maintained by themselves such as SO, SDC, AAI, etc. This increases the development and maintenance efforts

This introduces the other side effects of

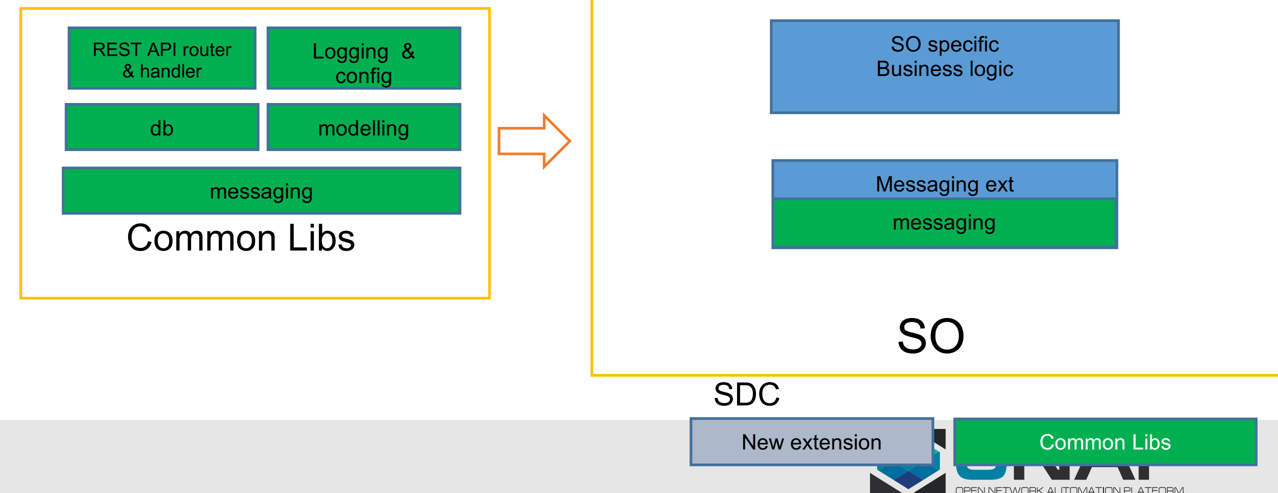
1. license violations,
2. security violations.
3. Sonar issues.
4. Code duplications
5. Duplicate efforts…



Proposed Solution:

✓ Introduce a common Lib/ SDK for the entire ONAP projects and make it avaialable for the extension of the projects.

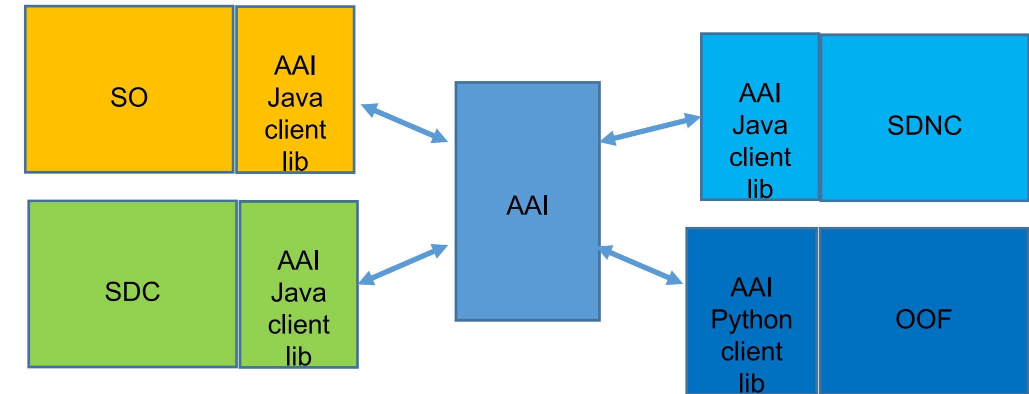
✓ This would leverage the projects in concentrating on their business specific functional logics and the other aspects would be solved all in once.



Unified Programming interface

Problem statement

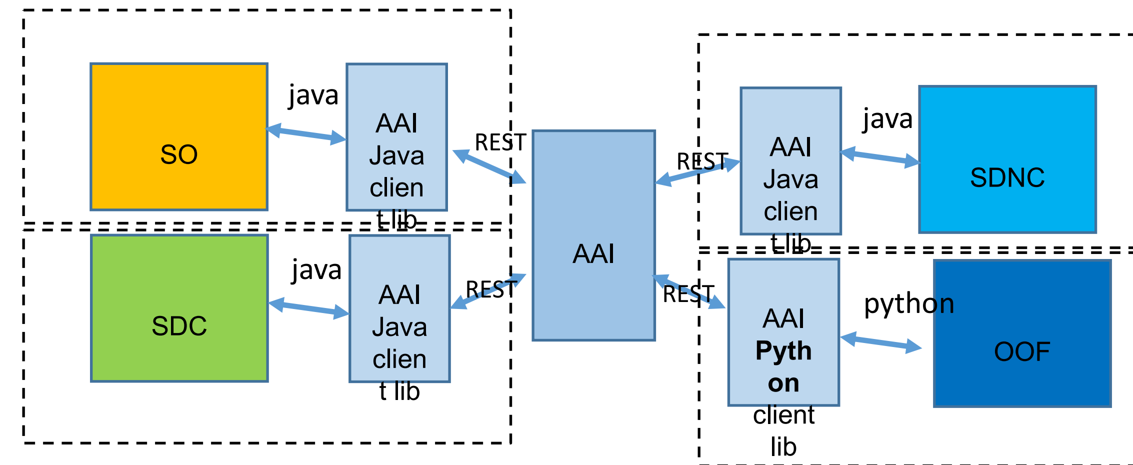
- In ONAP, every micro-service provides their own swagger document in json format, and when one service wants to integrate with another service, It implements the language specific client sdk and integrates.
- For example, Currently SO, SDNC, SDC, OOF, Holmes are having the own implementation of AAI client sdk.
- Same applicable for DMaaP.



Solution

- ✓ This problem could be solved by making the respective services to implement the client SDKs and whenever there is a change in API, corresponding service could update the SDK and deliver it.
- ✓ To automate this process, In ONAP, swagger-sdk component would help in auto-generate the push the new client SDK into nexus. So other services which is consuming could directly use it.

More details: https://wiki.onap.org/display/DW/swagger-sdk



Modularity within SO

- SO has started as a monolithic process handling multiple functional aspects within one docker component.

- In Short SO monolithic and large, not sharing common utilities

- This has created issues at different levels like the
  - Tight coupling between the modules
  - Hard to maintain as the functional aspects grow big and bulky and are relatively harder to maintain.
  - Deployment issues due to the build time for the entire component to be ready

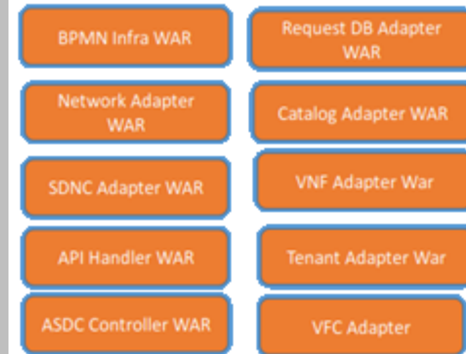
How to Solve:

- ✓ The current problem is big enough to be solved all at once and should be taken in incremental approach.
- ✓ First to all bring in the modularity by breaking the current SO into multiple smaller functional blocks
- ✓ Break SO to small, single-capability focused, standalone services
- ✓ Make the functional blocks re-usable across ONAP by providing a standardized APIs
- ✓ Bring-in plug and play capability esp for the adapters to have them easy to integrate with other components (that could even be external to ONAP)
- ✓ Implement a business capability accessed through a defined set of APIs
- ✓ Make the functional aspects more re-usable across the ONAP and give the capability for any component to call SO for its fulfillment. Without the need to re-write the logic required
- ✓ Make the SO components Scalable, distributable

Modularity in ONAP

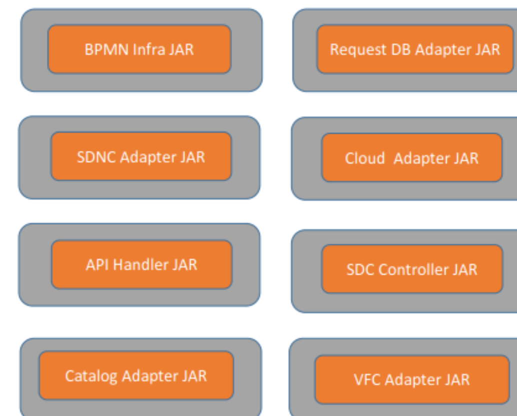
Beijing Deployment Scenario

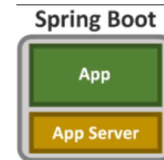
- Single webserver based
- Widlfy application server based
- Individual WAR deployments
- Scale of each WAR effects another
- Entire Java EE Stack



New Code deployment scenario

- Base Image uses Alpine Linux
- Replace Wildfly with Tomcat
- Individual Docker images
- Individual deployment/scaling capability
- Utilizing SpringBoot Stack



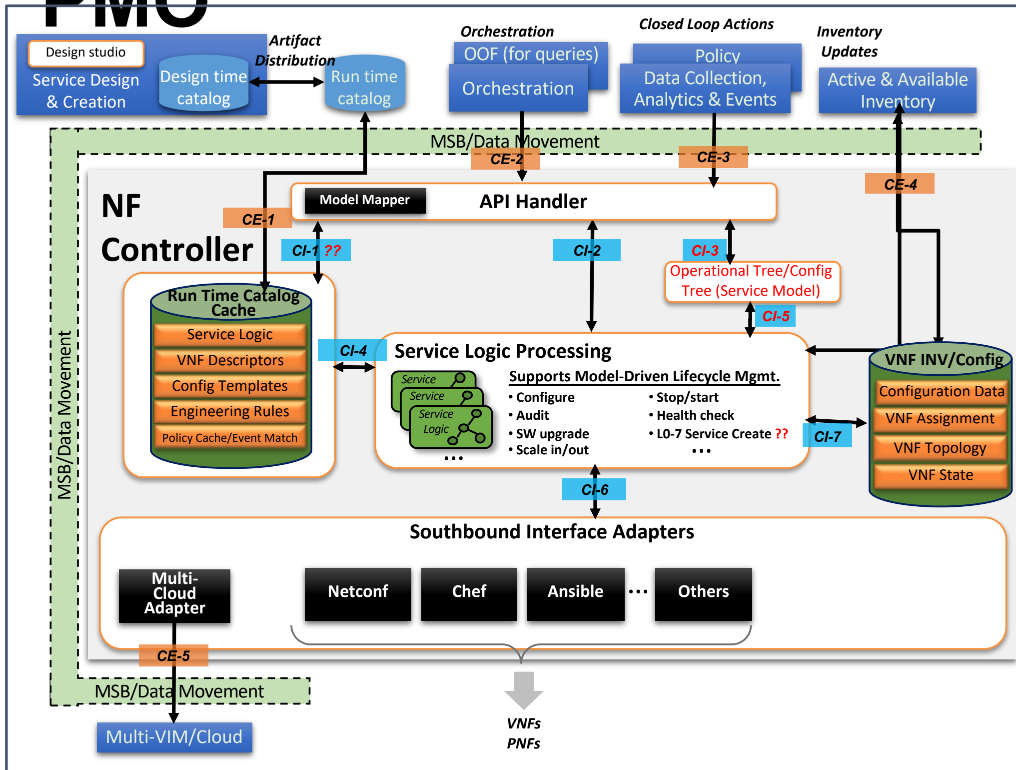


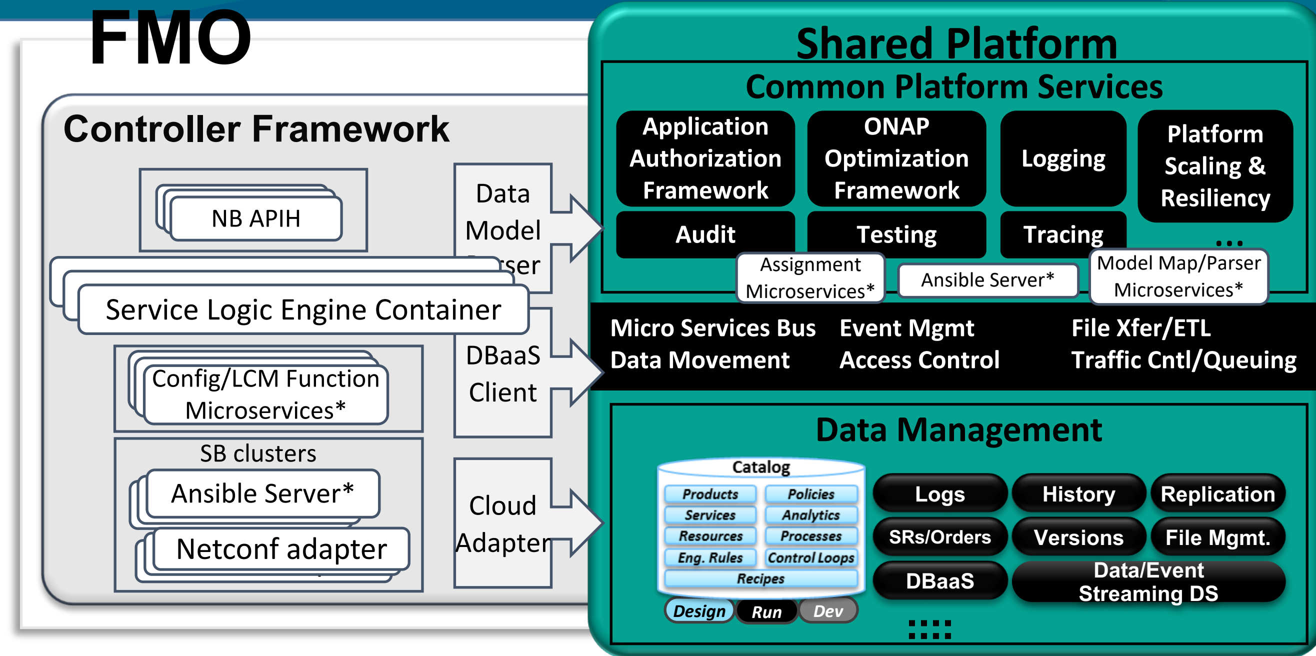
SO in Dublin

- This work effort will be targeted for Dublin and would cover the following:
  - SO would be decomposed into functional areas – as a target for the Dublin release (e.g.):
    - API handler
    - Request DB
    - BPMN Infra
    - SDC controller
    - Catalog Adapter
    - Adapters for the controllers (SDNC/VFC/…) and
    - Cloud Adapter
  - Each functional area would maintain the current API technology when containerized (REST->REST, RPC→RPC)
  - Any functional areas that don’t support either REST or RPC in their current state would be differed to R5
  - Unless otherwise noted, no new functionalities would be added to the new Microservice(s)
  - This work effort would be further scoped and T&C’ed by the respective PTL’s of other module

ONAP Controller functional decomposition plan

Controller: Targeted Improvements





- Extend and expand use of shared platform: AAF, Logging, DMaaP, …
- Common logging, audits and tracing: Platform-wide analytics
- Scaling and Resiliency through platform features (e.g. Kubernetes)
- DBaaS: Use common DB instead of today's component DB
- Runtime catalog: Avoid caching copy as today
- Decouple from ODL where needed
- *Evolve to autonomous microservices
  - Some shared across controller personas
  - Some as common services, consumed by any component (e.g., ansible)
  - Scalable independently

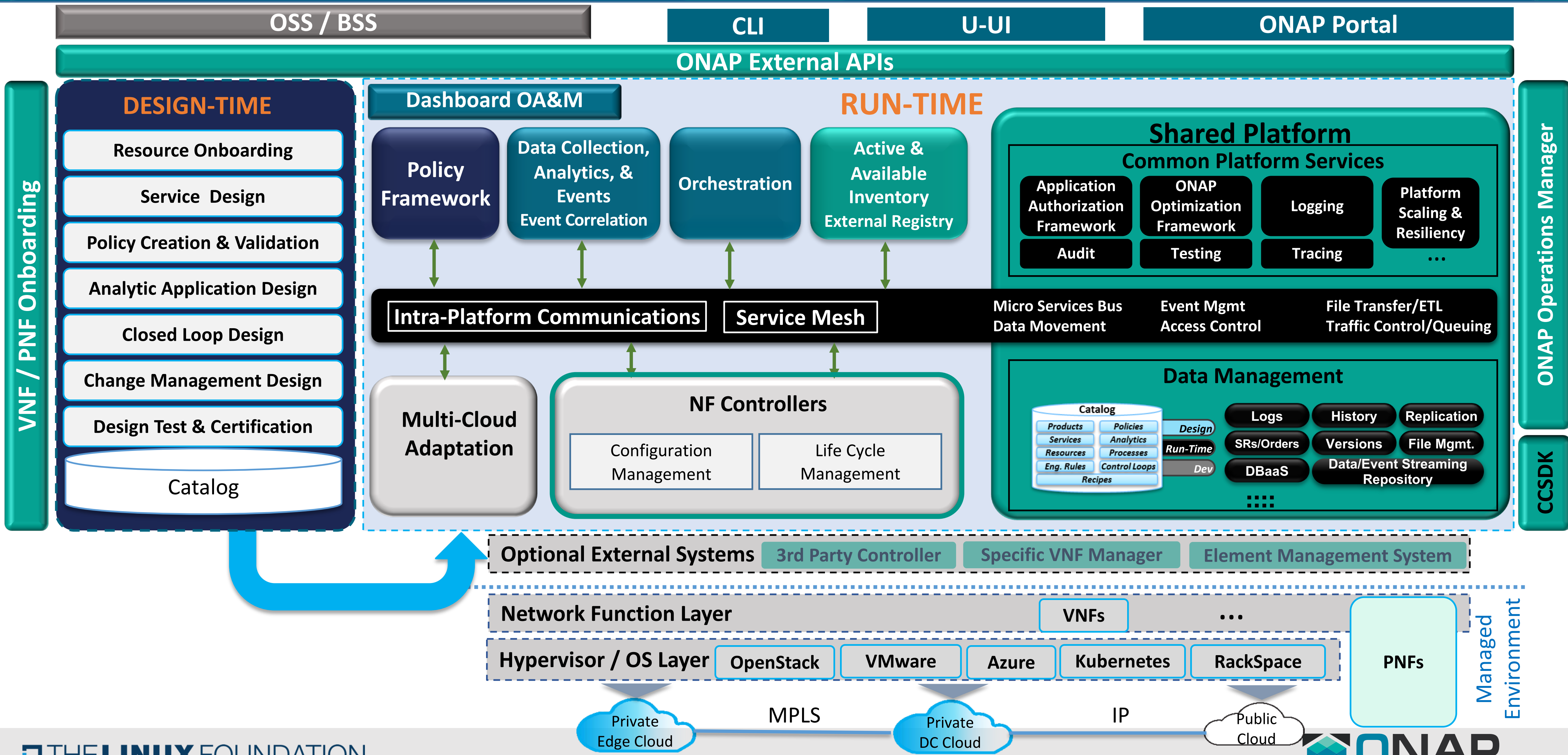
Controller Refactoring Example

Refactor controller to focus on SL execution, delegate common services/Data Mgt to the Shared Platform layer

Modules	Controller (PMO)	Controller Framework (FMO)	Goals Achieved
Run Time Catalog Cache	Controller	Platform: Data Mgt., Controller: DBaaS client	Reduce footprint of Component
Data Store	Controller –MySQL	Platform: Data Mgt., Controller: DBaaS client	Eliminate DB duplication; unify data management
Model Mapper/Parser (yang, tosca)	Controller	Platform Model Parser/Mapper App	Single reusable parser set – no duplicity
Other Utilities	Controller	Platform – audit, history, logging …	Relies on platform services & Reduces Dev $
Cloud API	Controller	Controller – adapter container	Reuse multi-cloud for all cloud/container infra
NB API Handler	Controller	Controller NB REST adapter	Consolidated API adapter across platform
SB adapters (yang/nc, ansible ..)	Controller/ODL	Platform common service or Controller level containers	Consolidated API adapter across platform & reuse platform services
Operational/Config Tree	Controller ODL	Platform: Data Mgt., Controller: DBaaS client	Eliminate DB duplication and redundancy
Karaf bundle – service logic (java)	Controller ODL	Controller microservices	Scalable, reusable, modular m-services
Resiliency & Scalability	Active-passive	Platform - dynamic on-demand scaling	Consistent platform scaling for all modules

Evolved ONAP Architecture

Refactor ONAP components as microservices, and build out Common Platform Services and Data Management



Conclusion and Next Steps

Evolve ONAP to a more modular, agile architecture

Goals:

- Breaking ONAP components into smaller reusable modules
- Enabling technology swap-out for modules
- Reducing software footprint
- Allowing integration of non-ONAP components

Next Steps for Dublin:

- Extract IP assignment from the controllers as a common microservice
- Extract Tosca Parser from SO and make a common microservice
- Thoughts and Comments?