ONAP Modularization

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

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

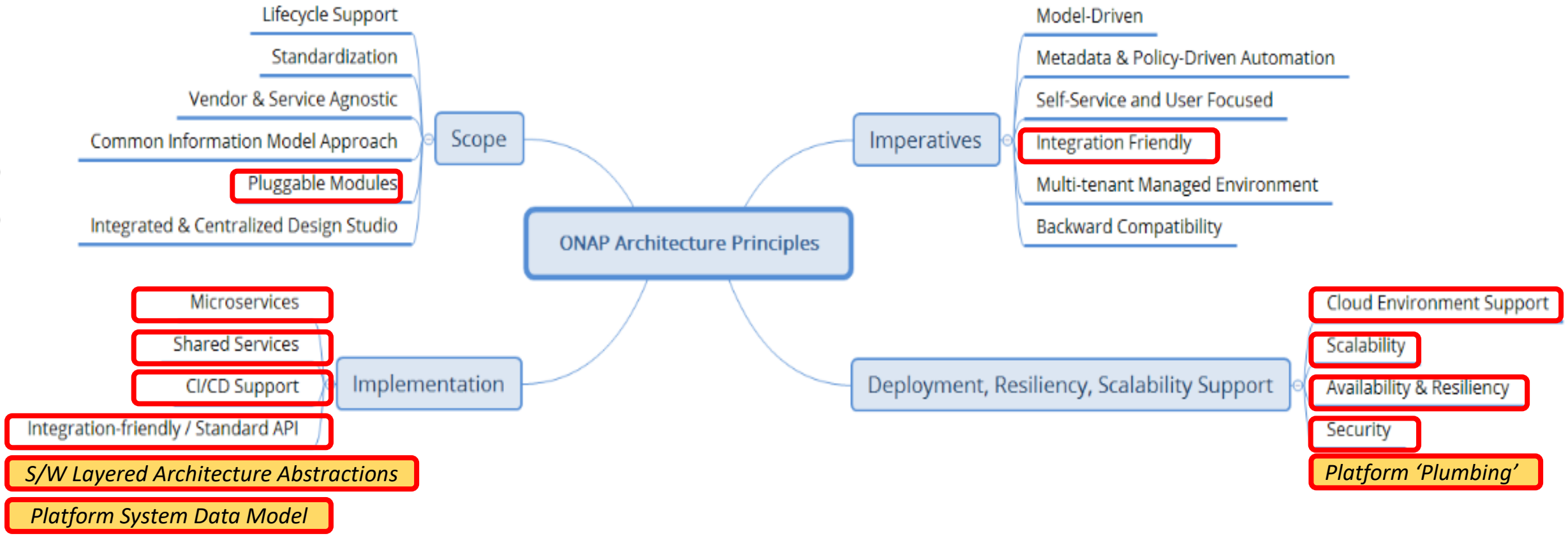
- Key ONAP challenges and critical gaps
  - Issues identified by the broader ONAP user community
- Architecture principles and approaches
  - Guide how to address the challenges
- Articulate succinct definitions and applicability of microservices, cloud-native, service mesh
- Refactor ONAP by leveraging common services to the fullest extent possible
- Approach: Focus on one major ONAP component at a time
  - Refactor/re-architect component to address specific challenges
- Example: ONAP controller: …. Coming soon
  - Challenges, suggested steps, and phased realization plan

Problem Statement

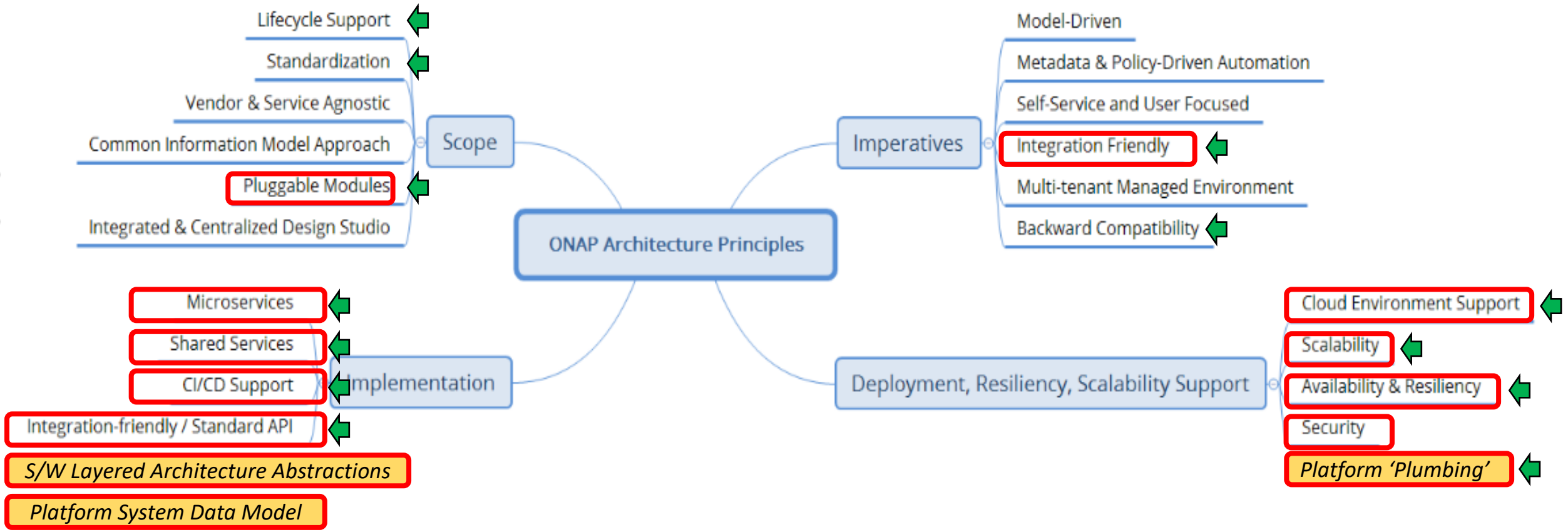
- There is a general perception that ONAP is too complex, too big and hard to make changes.
- Modules are monolithic (SDN-C, SO) and large, not sharing common utilities
- Service providers might have a specific module already implement and would like to integrate that module into ONAP (e.g. leveraging an external controller or orchestrator for some existing deployed technology)
- Service providers would like to deploy ONAP incrementally, whereas today ONAP supports all-or-nothing approach
- Not all ONAP modules take full advantage of cloud-native microservices

Can incorporate additional issues and/or more details if available

ONAP Architecture Principles



ONAP Architecture Principles



Legend: *Missing principle* | Principle not fully realized today | Proposal solution coverage

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

Create and Deploy Platform ‘Plumbing’

Areas of commonality

1. **Resiliency and Traffic Control**
   - Load balancing
   - Timeouts, Deadlines, Retry Budgets, Rate Limiting, Circuit Breaking
   - Recognizing and utilize idempotent behavior
   - Canary deployments, A/B tests
2. **Security**
   - Encryption decoupled from applications
   - Key rotation and certificate management (w Kubernetes)
3. **Observability**
   - Logging, auditing
   - Metrics
   - Distributed Tracing
4. **Data Persistence**
   - DBaaS
   - Configuration

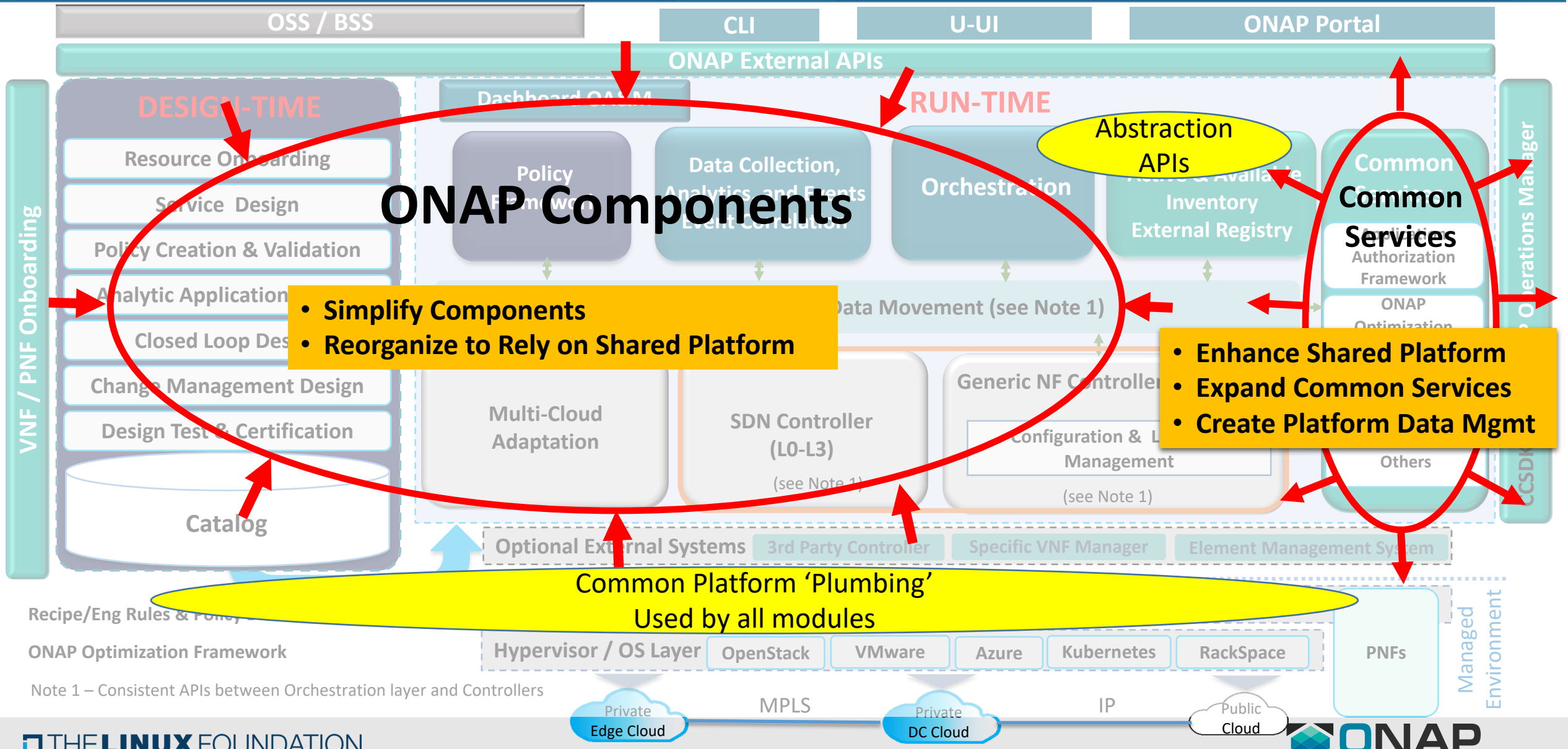
Toolings and Technologies

1. Microservices
2. Cloud-Native (e.g. CNCF)
   - Docker
   - Kubernetes
   - Service Meshes

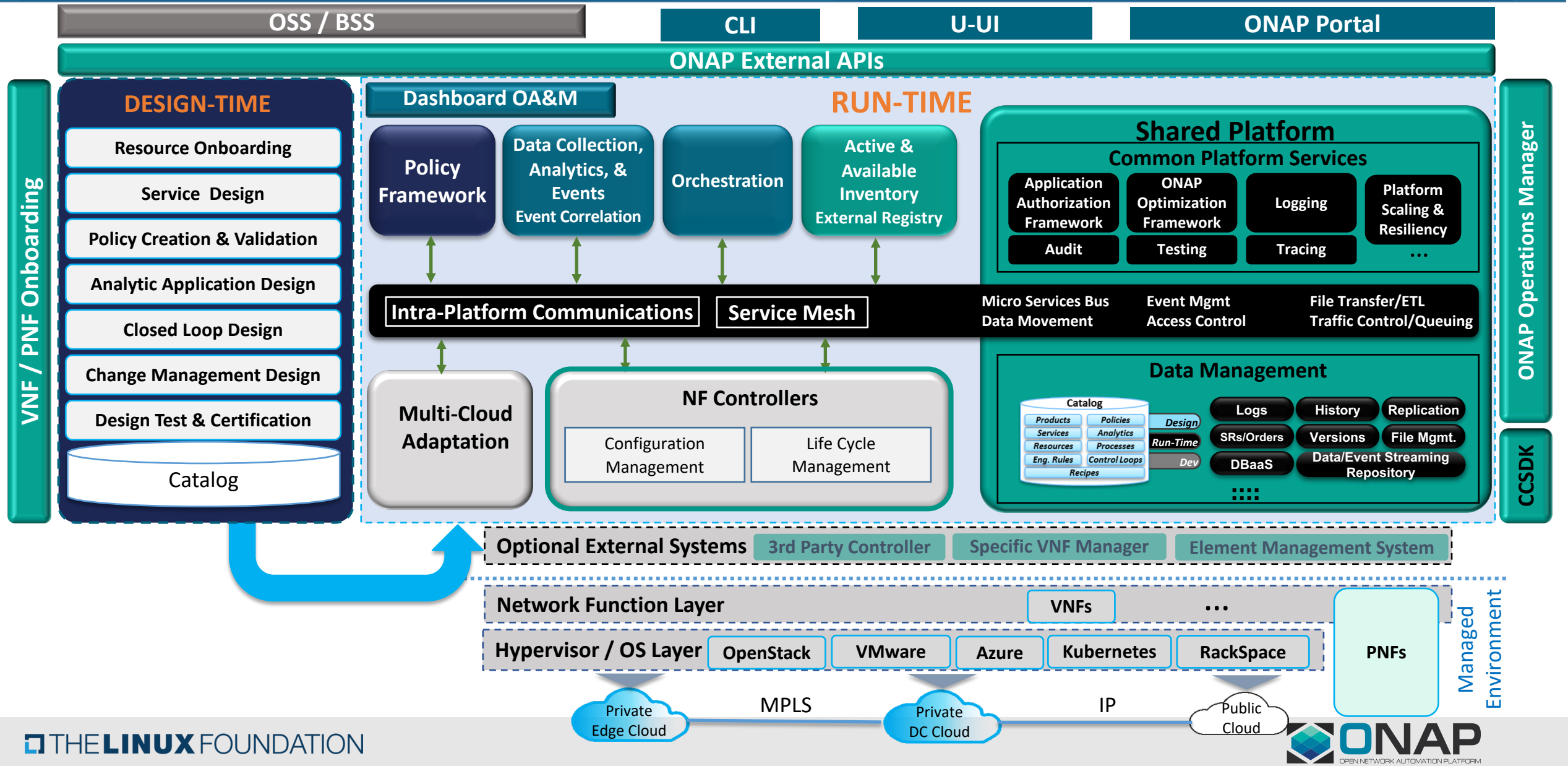
Contribute to General OSS efforts

- Large-scale networking support for Docker and Kubernetes
  - Who is better positioned than us to do this?

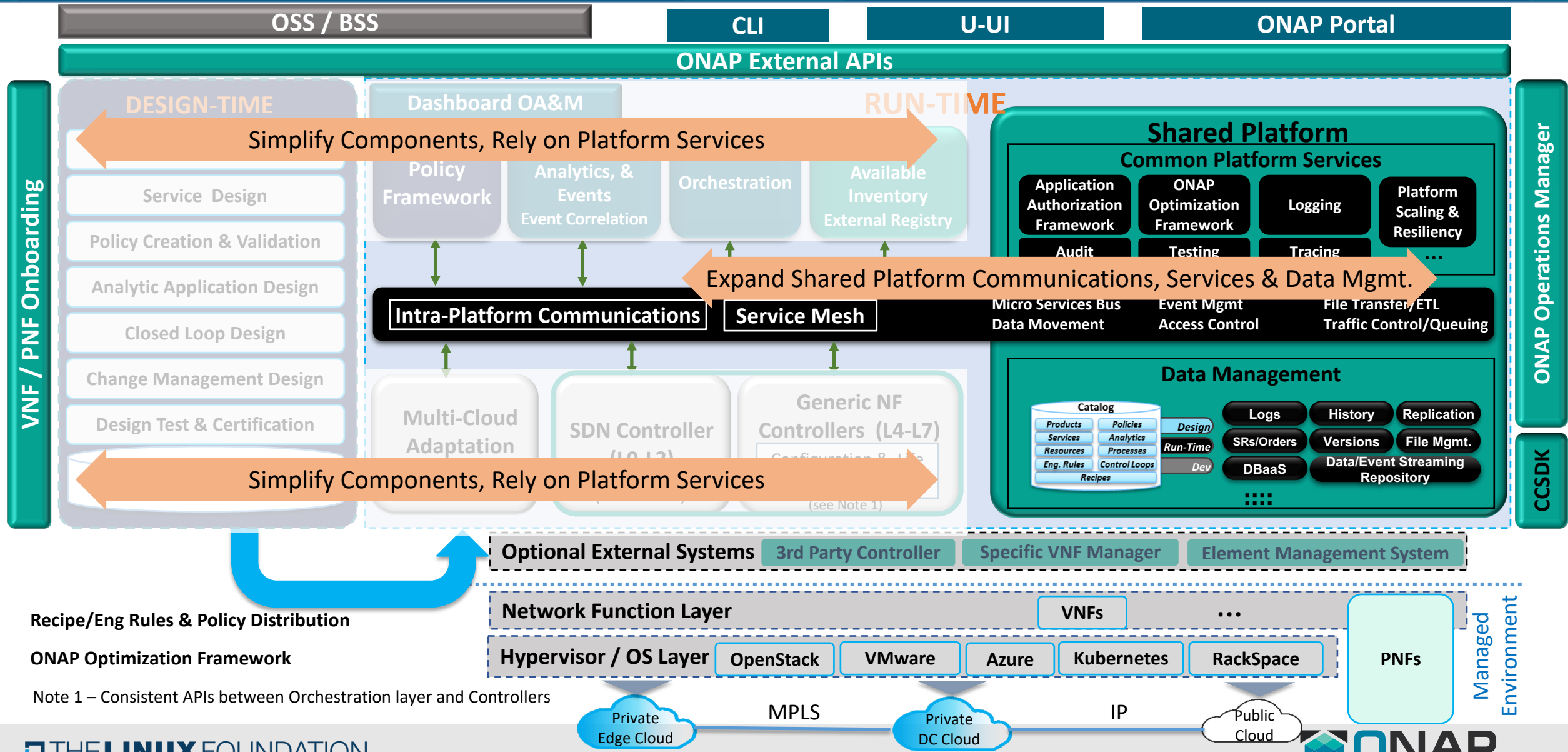
ONAP Architecture – Emphasis on Shared Platform Capabilities



Proposed ONAP Architecture Updates

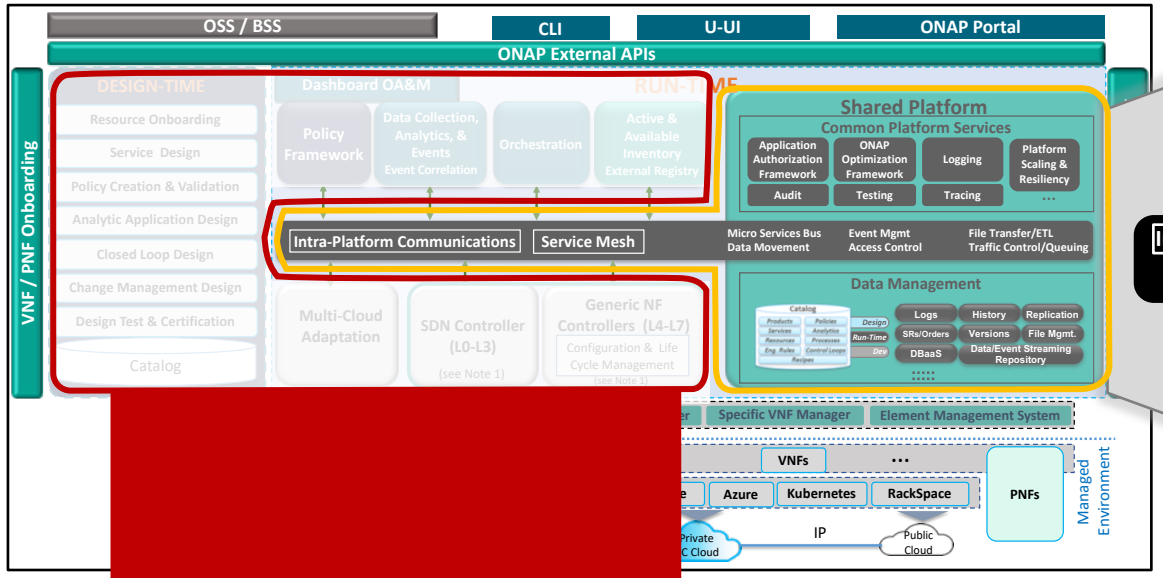


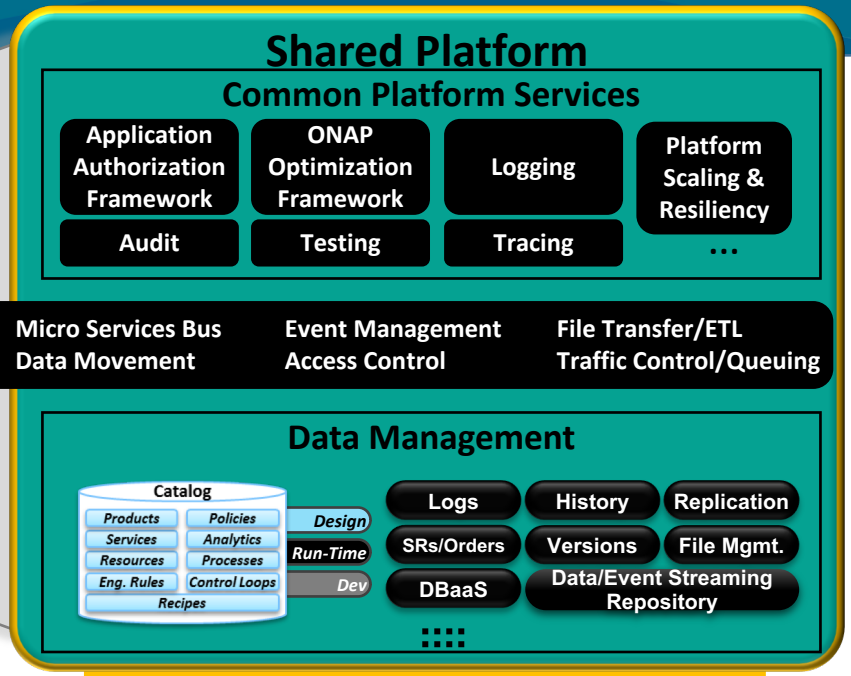
ONAP Architecture – Emphasis on Shared Platform Capabilities



Note 1 – Consistent APIs between Orchestration layer and Controllers

Shared Platform & Plumbing Supporting ONAP Components





ONAP Components

- **Functions**
  - Decompose toward modular, stateless microservices frameworks
  - Rely on Shared Platform Capabilities
  - Keep to leverage cloud native auto-heal/auto-scale
- **Characteristics**
  - Non-Duplicating
  - Consolidate and minimize footprint

Shared Platform

- **Functions**
  - Platform common services (shared utilities)
  - Resilience, failover, maintainability, operational consistency
  - Persistent Platform Data Management
- **Characteristics**
  - Must build first and must mature
  - Changes less often
  - View as a whole

In Summary

Goal: Evolve ONAP to a more modular, agile architecture

Breaking ONAP components into smaller reusable modules

- Decompose ONAP on a component-by-component basis
- Tie directly to addressing current problems
- Validation of approach

Enabling technology swap-out for modules

- Define abstract interfaces between modules
- Provide ability to change implementation over time
- Support partial use of ONAP component

Reducing software footprint

- Extract common services into a reusable platform
- Leverage technologies to support evolution (microservices, cloud-native, service mesh)

Allowing integration of non-ONAP components

- Support partial use of ONAP component
- Documented interfaces define integration points

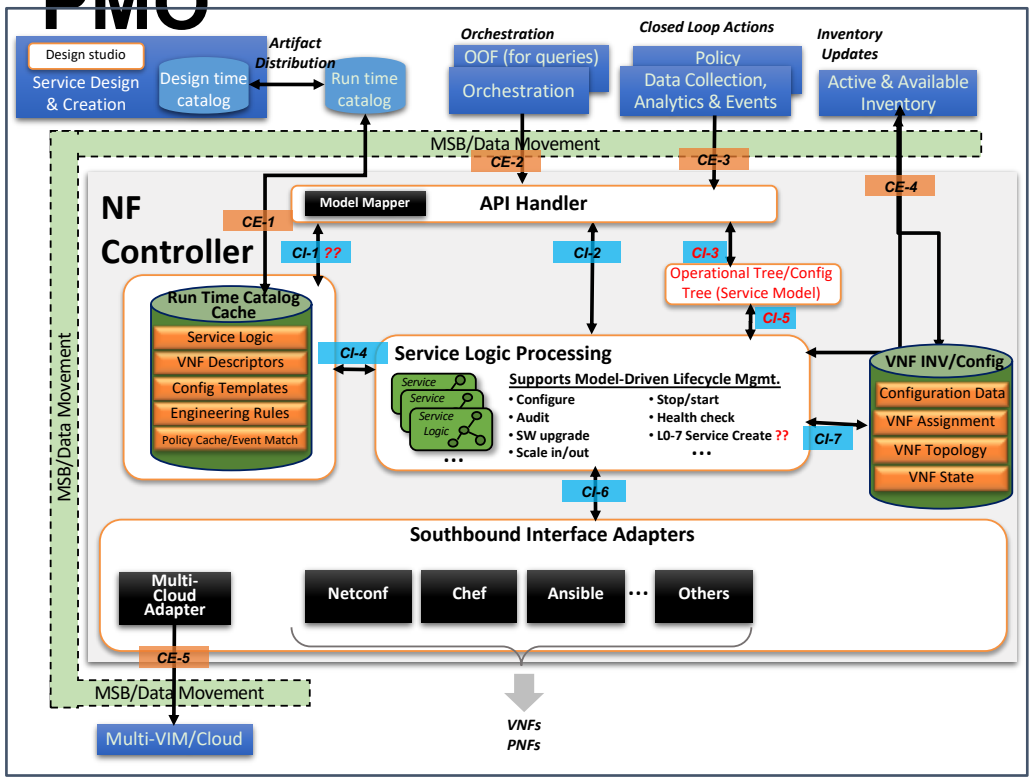
Example Component Modularization

Controllers: Current Issues and Challenges

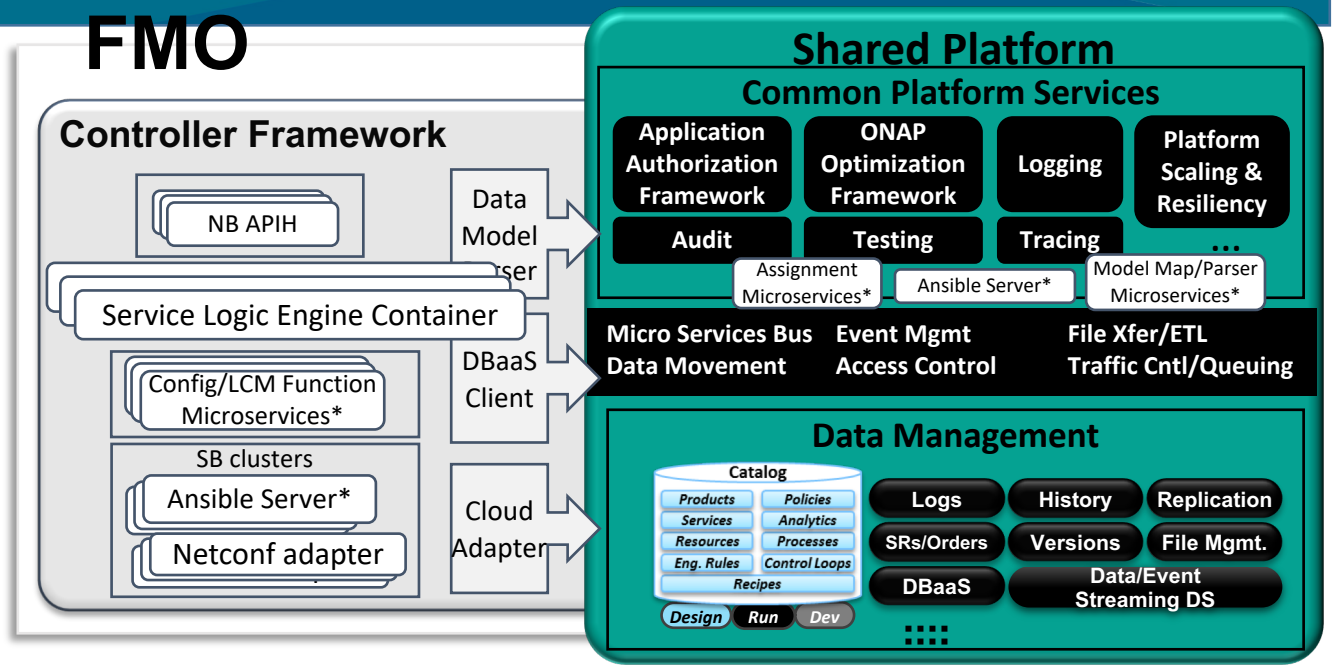
- Lack of clarity & roles in the controllers (which controller does what?)
- Divergence of controller implementation
- Duplicate and uncoordinated interfaces
  - Lack of full Configuration & Lifecycle Management by one controller
  - Lack of uniform common services in southbound interfaces
- Tightly coupled to Open Daylight (ODL) could affect modularity and technology refresh
- Controller scalability (functional and non-functional)
  - Functional: Instance(s) for each separate functionality?
  - Non-functional: Scalability due to transaction volume and load
- Identify and migrate to common modular services used by multiple components
  - Examples: IP Address Assignment, TOSCA Parser, YANG Parser, Ansible server
- Model-driven architecture not fully implemented

Controller: Targeted Improvements

PMO



FMO



- 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: Benefits of Suggested Work

Functionality

- Clarified roles and responsibilities
  - Necessary pre-req for modularization
- Consistency of common functionality
  - Implemented via shared modules & components
- Standardize and abstract common interfaces
  - Modularization supports loosely coupled services for easier swap-ins and swap-outs
- Scalability and resilience improved
  - Via use of shared platform common services
- Separation of Application Layer from Data Layer
- Extend and use common platform capabilities
  - DBaaS for Data Store
  - Runtime catalog instead of catalog cache
  - Yang and Tosca common parsers
  - Ansible servers

Principles Realized / *Enhanced*

- Scope
  - Pluggable Modules
- Imperatives
  - Integration Friendly
- Deployment, Resiliency, Scalability
  - *Scalability*
  - *Availability and Resiliency*
  - *Security*
  - Platform plumbing (New)
- Implementation
  - *Shared Services*
  - Microservices evolvable independently
  - Integration-Friendly/Standard APIs
  - Software Layered Architecture Abstractions (New)

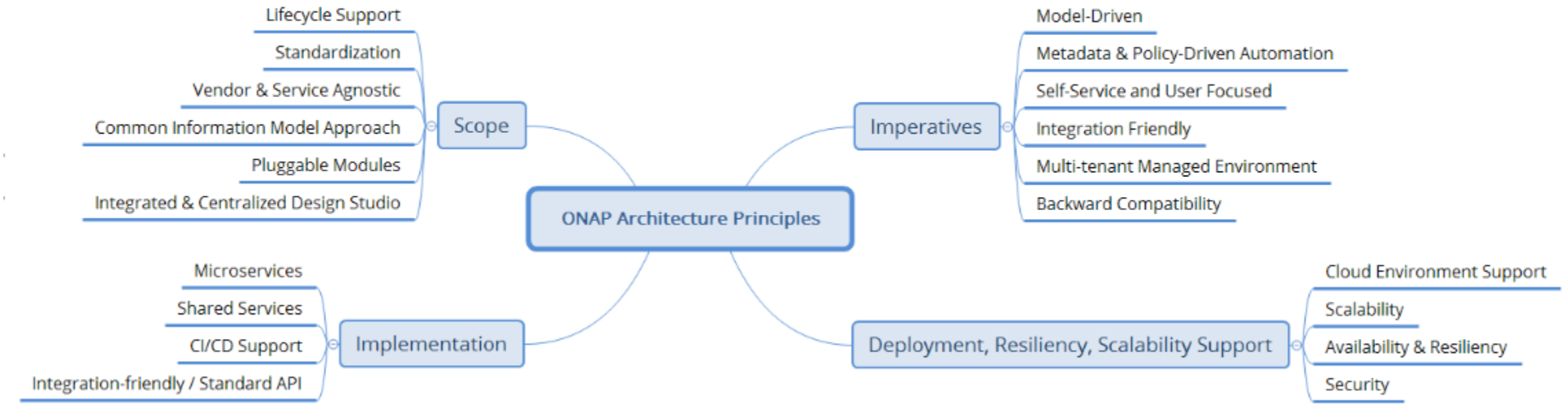
Controller Refactoring Example

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

ONAP Architecture Principles

ONAP Architecture Principles



ONAP Architecture Principles: Scope

- **Lifecycle Support**: ONAP must support a complete life cycle management of software-defined network functions / services: from VNF On-Boarding, Service Definition, VNF/Service Instantiation, Monitoring, Upgrade, to retirement
- **Standardization:** ONAP must support a common approach to manage various network functions from different vendors
  - Standard templates for instantiations
  - Standard language for configuration
  - Standard telemetry for monitoring and management
- **Vendor & Service Agnostic**: ONAP Platform must be VNF, Resources, Products, and Service agnostic. Each service provider or integrator that uses ONAP can manage their specific environment (Resources, VNFs, Products, and services) by creating necessary meta-data / artifacts using Design Studio to support their needs / environment.
- **Common Information Model approach:** ONAP should define a standardized common information model for all vendors to follow. This will allow ONAP users to quickly onboard and support new VNFs.
- **Pluggable Modules**: The ONAP architecture should develop and promote VNF standards to allow delivery of Lego block-like pluggable modules, with standard interfaces for all aspects of lifecycle management (e.g. instantiation, configuration, telemetry collection, etc.).
  - Provide common tooling that can be used by microservices to support standard functionality for resiliency, traffic control, observability and security (viz. service meshes)
- **Integrated & Centralized Design Studio**: All artifacts required for ONAP components should be able to be designed from a central ONAP design studio.

ONAP Architecture Principles: Business Imperatives

- **Automation:** ONAP must support Automation at every phase of Lifecycle
- **Model Driven**: All ONAP modules should be model-driven, avoiding, where possible, programming code. This allows for a catalog-based reusable repository for network & services lifecycle management.
- **Meta-data & Policy Driven Automation**: ONAP should support high levels of automation at every phase of lifecycle management – e.g. onboard, design, deployment, instantiation, upgrade, monitoring, management, to end of life cycle. These automations should be policy driven, allowing users to dynamically control automation behavior via policy changes.
- **Self-Service & User Focused**: ONAP Platform should support a self-service model with a fully integrated user-friendly design studio to design all facets of lifecycle management (product/ service design, operational automation, etc.). All interfaces and interactions with ONAP should be user friendly and easy to use.
- **Integration Friendly**: When an ONAP component relies on software outside of the ONAP project, the dependency on that external software should be designed to pluggable, API-oriented, supporting multiple possible implementations of that dependency.
  - Use of microservices will of necessity document interfaces to be supported and provide reference implementations, both of which will make creation of alternate versions easier
- **Multi-tenancy managed environment:** The ONAP platform should support the ability manage multiple tenants and provide isolation for those tenants.
- **Backward Compatibility**: ONAP platform should support backward compatibility with every new release.

ONAP Architecture Principles: Implementation Approach

- **Microservices**: ONAP modules should be designed as microservices: service-based with clear, concise function addressed by each service with loose coupling.
  - Support extensive use of microservices as a means of supporting (a) loosely-coupled agile development, test, and deployments, (b) runtime scalability, resilience, and decreased footprint, and (c) feature reusability
- **Shared Services**: Where applicable, reusable components can be provided as shared services across ONAP components.
  - Build out the common services as part of a shared platform, supporting (a) common platform services (e.g. security), (b) data management (e.g. logging), and (c) intra-platform communications (e.g. traffic controls and tracing)
- **CI / CD Support**: ONAP is predicated on an accelerated lifecycle for network services. As such, agility is key in all aspects of ONAP: development of ONAP, designing of network services, and operation of both ONAP and network services. Principles of continuous integration and deployment should be followed by all modules of the ONAP platform.
  - Extend and support current CI/CD standards and implementations to support agile development and deployment of individual microservices and introduction of same in a controlled manner
- **Integration Friendly / Standard API**: Various service providers and users of ONAP should be able quickly integrate ONAP with their existing OSS / BSS systems. An open, standards-based architecture with well-defined APIs fosters interoperability both within ONAP and across complementary projects and applications
  - Use of microservices will of necessity document interfaces to be supported and provide reference implementations, both of which will make creation of alternate versions easier. Integration with non-ONAP components similarly facilitated by defining all ONAP APIs interfacing externally.
- **Software Layered Architecture Abstractions:** Define ONAP as a layered architecture similar to the OSI model for the internet. Define abstract interfaces between the different layers to support information and request flowing between the layers in an implementation-independent manner.
- **Platform System Data Model:** Defines an abstract data model of the objects and entities to be managed by ONAP.

ONAP Architecture Principles: Deployment / Resiliency / Scalability Support

- **Cloud Environment Support**: All components in ONAP should be virtualized, preferably with support for both virtual machines and containers. All components should be software-based with no requirement on a specific hardware platform.
  - Component refactoring to address current problems will extend the use of microservices, service meshes and cloud-native technologies, moving to a fuller implementation of this principle
- **Scalability**: ONAP must be able to manage a small set of VNFs to highly distributed, very large network and service environment deployed across the globe.
  - Most refactored microservices will be ephemeral, supporting scalability through dynamic instance creation, movement, restart, and teardown
- **Availability & Resiliency**: ONAP must support various deployment and configuration options to meet varying availability and resiliency needs of various service providers.
  - Most refactored microservices will be ephemeral, supporting scalability through dynamic instance creation, movement, restart, and teardown
- **Security:** All ONAP components should keep security considerations at the fore-front of all architectural decisions. Security should be a pervasive underlying theme in all aspects of ONAP**.** The ONAP architecture should have a flexible security framework, allowing ONAP platform users to meet their security requirements.
  - Enhance security by decoupling development from the components and modules by providing features as a common service, sidecar or pluggable module. Support enhanced & automated key rotation and certificate management.
- **Platform Plumbing:** Identifies areas of commonality and implements reusable solutions that can be used to support generic needs such as (a) resiliency and traffic control, (b) observability, (c) security, and (d) data persistence, alleviating the burden of this on the module developers, and speeding up the process accordingly.

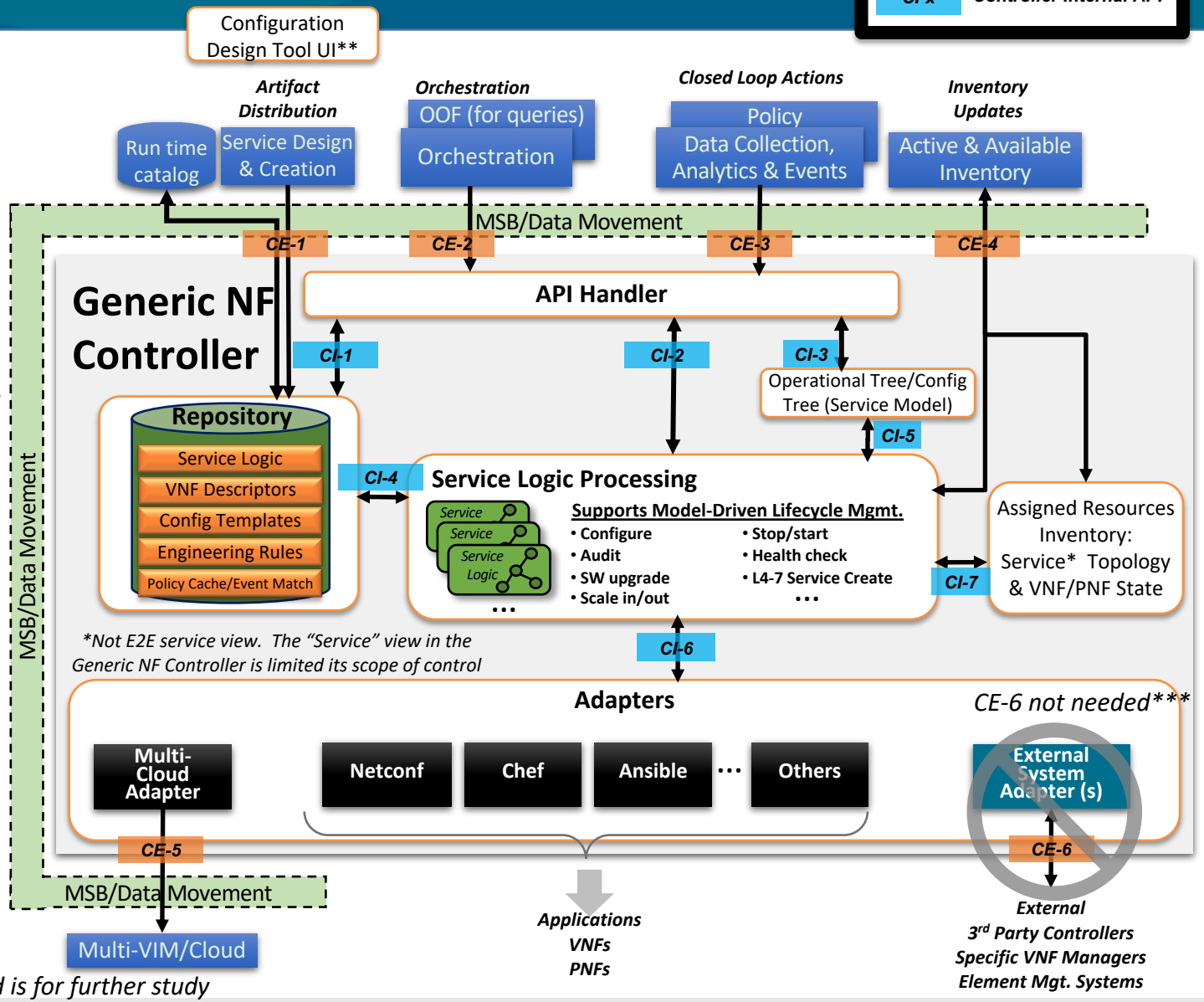
Note: The following Controller Related Slides are pulled from ONAP R4+ Architecture Tiger Team Report (to ARC) deck as reference:
https://wiki.onap.org/download/attachments/16003414/py20181011%20ONAP%20Modularization%20Considerations.pptx?version=4&modificationDate=1539624673000&api=v2

R4+ Architecture Slides

Last Update - July 30, 2018

Generic NF Controller Architecture

- **Generic NF Controller configures and maintains the health of VNFs/PNFs/services* (L1-7) throughout their lifecycle.**
  - **The Lifecycle Management Functions are a normalization of the controller aspects of VF-C and APP-C functions into a common, extensible library**
- **Programmable network application management platform**
  - **Behavior patterns programmed via models and policies**
  - **Standards based models & protocols for multi-vendor implementation**
  - **Extensible SB adapter set including vendor specific VNF-Managers**
  - **Operational control, version management, software updates, etc.**
- **Manages the health of VNFs/PNFs within its scope**
  - **Policy-based optimization to meet SLAs**
  - **Event-based control loop automation to solve local issues near real-time**
- **Local source of truth**
  - **Manages inventory within its scope**
  - **All stages/states of lifecycle**
  - **Configuration audits**
- **Key Attributes of Generic NF Controllers**
  - **Intimate with network protocols**
  - **Manages the state of services**
  - **Provide Deployment Flexibility to meet user scalability / resilience needs**

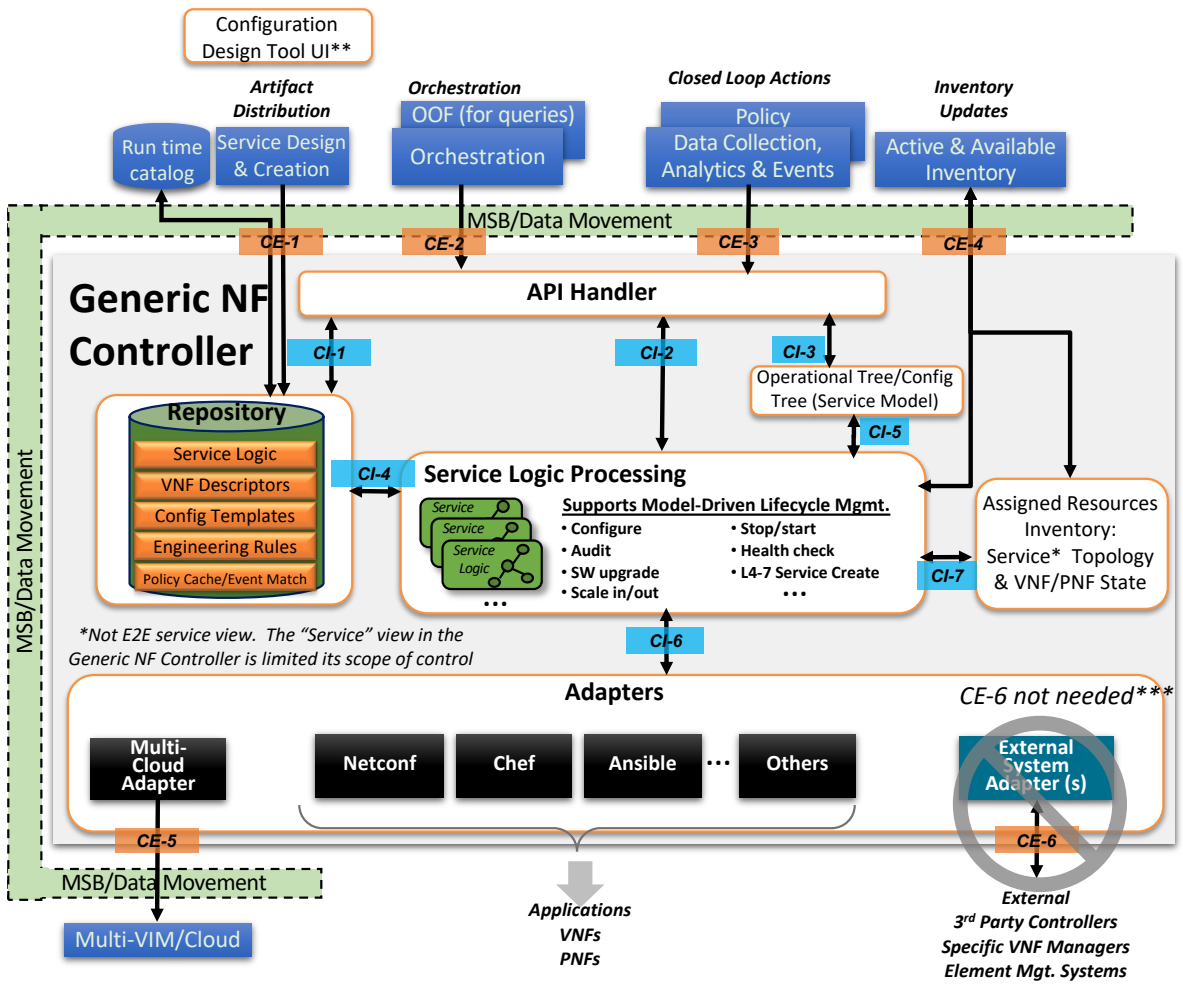


*How the services are to be handled is for further study

** Configuration Design Tool (CDT) to be integrated into SDC

***CE-6 not needed - see External Controller materials

Generic NF Controller – External/Internal Interface Definitions



	Interface Definitions
CE-1	Distribution of artifacts from Service Design and Creation – artifacts distributed to Run Time Catalog, GNFC receives notification and pulls from Run Time Catalog *Note: Configuration Design Tool UI to be integrated into Service Design & Creation*
CE-2	Service requests from Orchestration ONAP Optimization Framework (OOF) queries for VNF state and available capacity
CE-3	Closed Loop action requests from Data Collection, Analytics & Events/Policy
CE-4	Inventory retrieval from Active & Available Inventory by Service Logic Processing engine Inventory updates to Active & Available Inventory by Assigned Resources Inv
CE-5	Lifecycle management requests to Multi-Cloud (e.g., stop/start VM)
~~CE-6~~	~~Lifecycle management requests to an external controller or system that has responsibility of the target VNF~~
CI-1	API Handler looks up or retrieves the corresponding Service Logic instance that maps to NB service request (service/network yang)
CI-2	API Handler calls Service Control Processing to perform the Service Logic on the target service or network
CI-3	Prior to CI-2, API Handler might query the (in-memory) Operational/Config Trees for the network or service details (if already existing)
CI-4	Service Control Processing retrieves the Service Logic, Config Templates, Engineering rules, and Policies as part of processing the requested action
CI-5	Service Control Processing queries and/or updates Operational/Config Trees as part of making changes to the network (VNFs/PNFs)
CI-6	Service Control Processing requests adapter layer to update/configure VNF/PNF update using the appropriate adapter for the VNF/PNF
CI-7	Service Control Processing queries and/or updates local Assigned Resources Store/Inventory as part of making changes to the network (VNFs/PNFs)

GNFC – External Interface Details

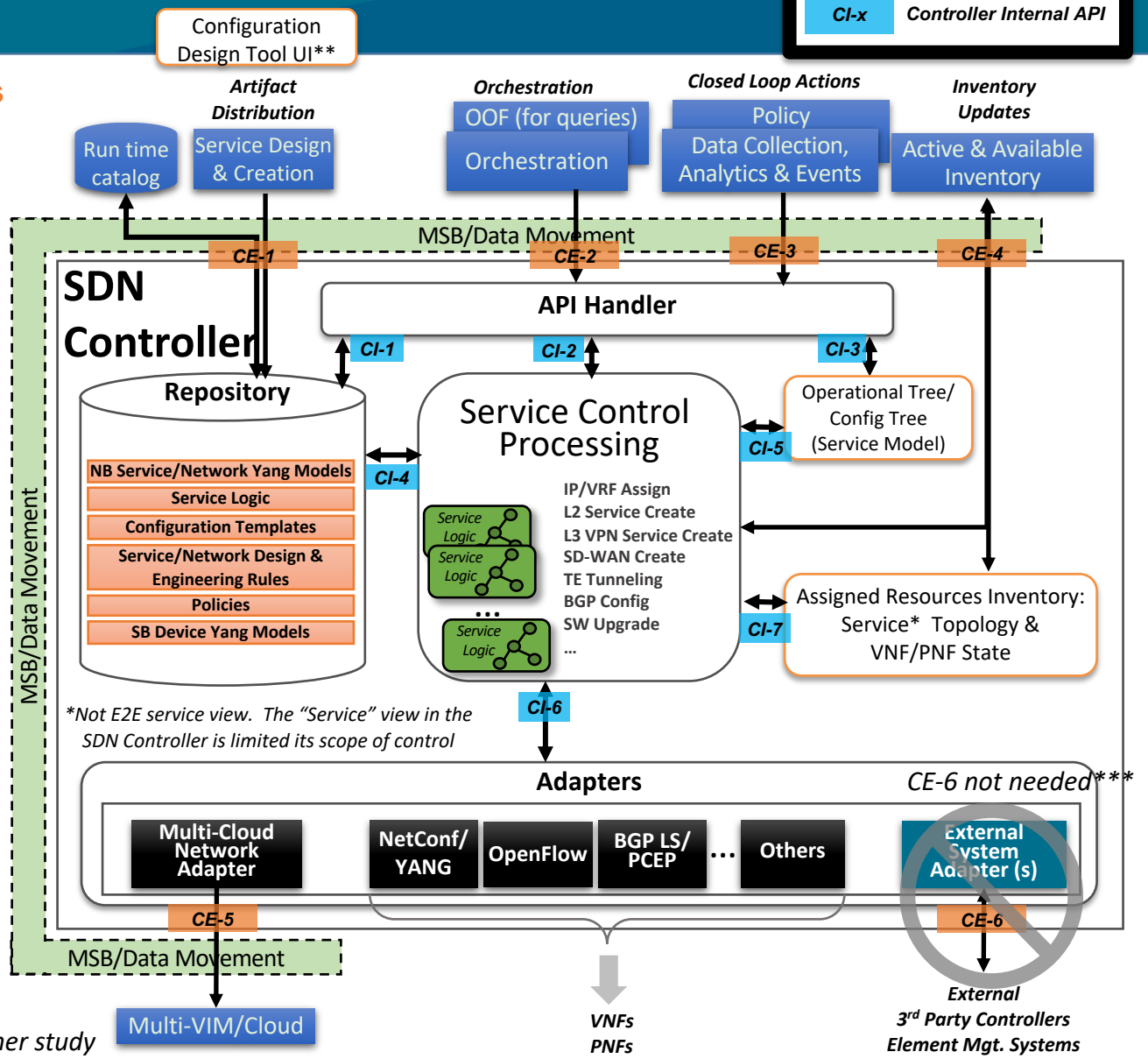
	Interface Definitions	Beijing Rel.	Casablanca Rel.	Protocol /Service	Comments
CE-1	Distribution of artifacts from Service Design and Creation	SDC➔[no GNFC]	SDC ➔ GNFC (trigger) GNFC ➔ Run Time Catalog (pull)	DMaaP	
CE-2	Service requests from Orchestration Queries from ONAP Optimization Framework (OOF) for VNF state and available capacity	SO, Portal ➔[no GNFC] OOF ➔ [no GNFC]	SO, Portal ➔ GNFC OOF queries – not in scope?	REST	Generic Request API. See next slide for orchestration requests for LCM actions.
CE-3	Closed Loop action requests from Data Collection, Analytics & Events & Policy	DCAE ➔ [no GNFC] Policy – not in scope	DCAE ➔ GNFC Policy – not in scope	DMaaP	
CE-4	Inventory retrieval from Active & Available Inventory by Service Logic Processing engine Inventory updates to Active & Available Inventory by Assigned Resources Inventory	A&AI ⇔ [no GNFC]	A&AI ⇔ GNFC	REST	
CE-5	Configuration requests for cloud infrastructure networking Lifecycle management requests to Multi-Cloud (e.g., stop/start VM)	Multi-Cloud – not in scope	GNFC ➔ M-Cloud	REST	

- Controllers are to be Model-Driven – APIs in Dev, Design, Run-Time catalogs
- Payloads: parameter values defined in the platform Data Dictionary (model/meta-data driven)
- CE-6 interface (to external controllers) is not needed and has been deleted. External controller will be interfacing to the whole ONAP platform – via CE-1 thru CE-4
- Beijing Release does not have an implementation of GNFC
- For Casablanca it is recommended that VF-C and APPC begin to transition toward GNFC

Note: Beyond Casablanca, SDNC functionality and software will begin to merge into GNFC

SDN-Controller Architecture

- **SDN Controller configures and maintains the health of VNFs/PNFs for cloud networking (underlay/overlay) and WAN transport services* throughout their lifecycle**
- **Programmable network application management platform**
  - **Behavior patterns programmed via models and policies**
  - **Standards based models & protocols for multi-vendor implementation**
  - **Extensible SB adapter set supporting various network config protocols, including 3rd party controllers**
  - **Operational control, coordinated state changes across devices, source of telemetry/events, etc.**
- **Manages the health of VNFs/PNFs/transport services in its scope**
  - **Policy-based optimization to meet SLAs**
  - **Event-based control loop automation to solve local issues near real-time**
  - **Action executor for outer control loop automation**
- **Local source of truth**
  - **Manages inventory within its scope**
  - **All stages/states of lifecycle**
  - **Configuration audits**
- **Key Attributes of Controllers**
  - *Intimate with network protocols*
  - *Manages the state of services*
  - *Single service/network domain scope per instance*



*How the services are to be handled is for further study

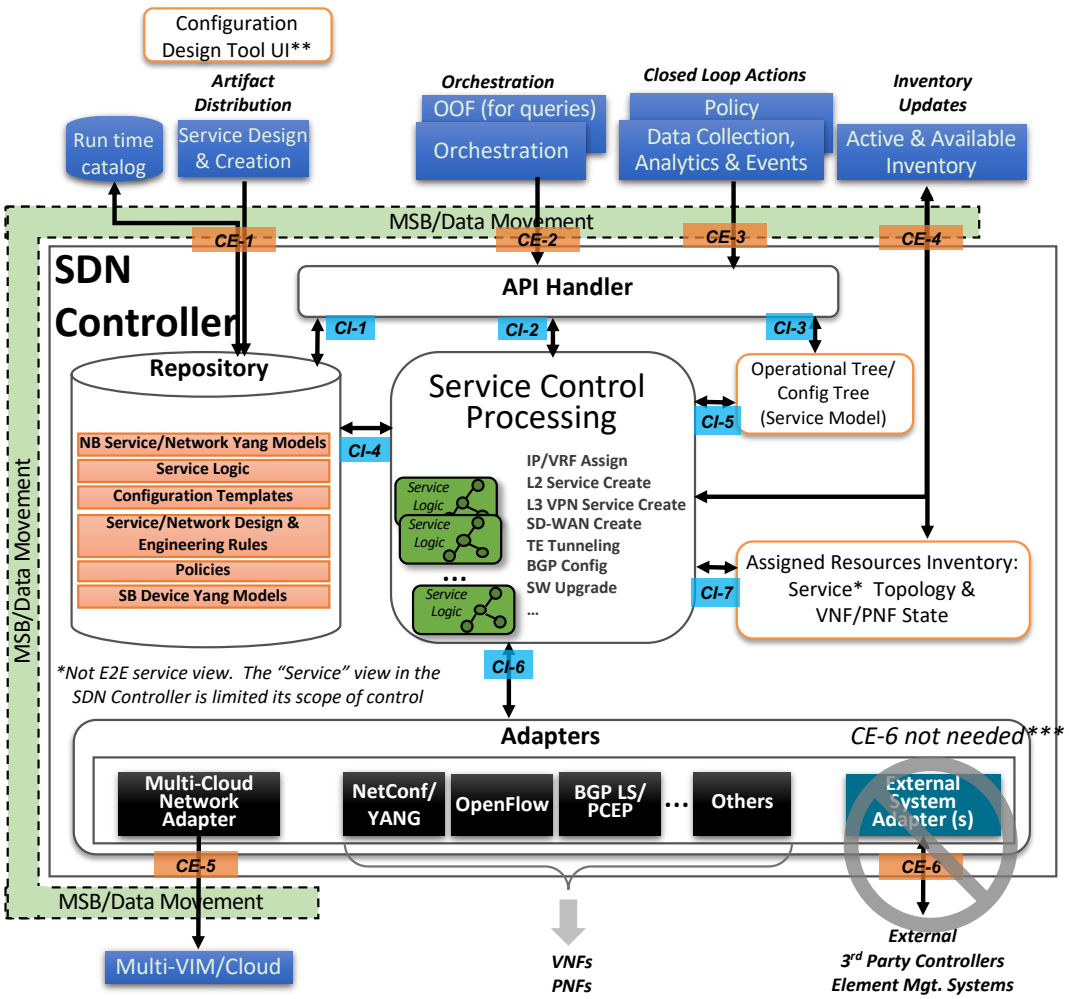
** Configuration Design Tool (CDT) to be integrated into SDC

***CE-6 not needed - see External Controller materials

SDN-Controller – External/Internal Interface Definitions

Key

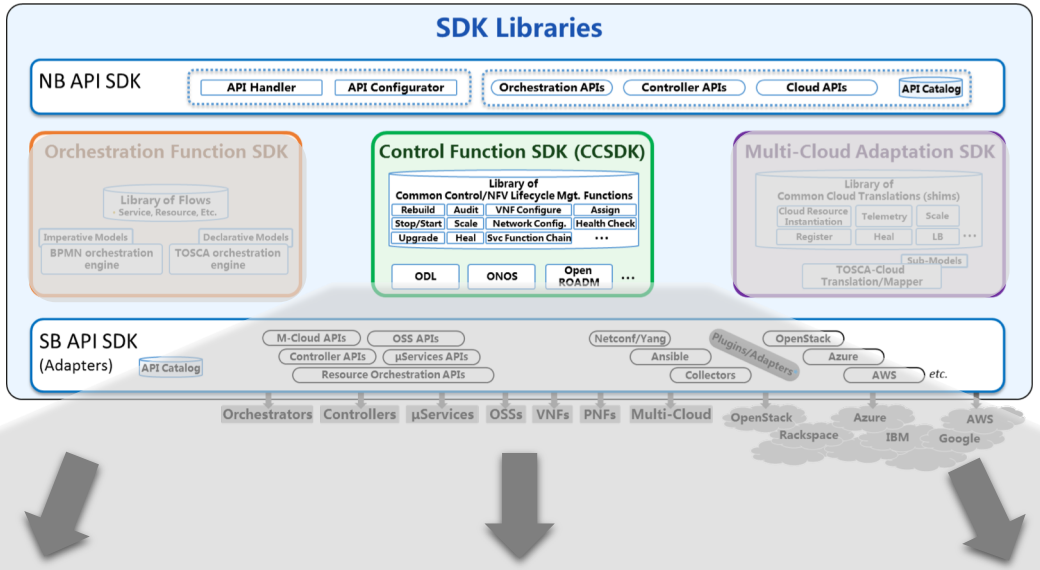
CE-x	Controller External API
CI-x	Controller Internal API



	Interface Definitions
CE-1	Distribution of artifacts from Service Design and Creation – artifacts distributed to Run Time Catalog, SDNC receives notification and pulls from Run Time Catalog *Note: Configuration Design Tool UI to be integrated into Service Design & Creation*
CE-2	Service requests from Orchestration Queries from ONAP Optimization Framework (OOF) for VNF state and available capacity
CE-3	Closed Loop action requests from Data Collection, Analytics & Events/Policy
CE-4	Inventory retrieval from Active & Available Inventory by Service Control Processing engine Inventory updates to Active & Available Inventory by Assigned Resources Inventory
CE-5	Configuration requests for cloud infrastructure networking Lifecycle management requests to Multi-Cloud (e.g., stop/start VM)
~~CE-6~~	~~Lifecycle management or configuration requests to an external controller or system that has responsibility of the target VNF~~
CI-1	API Handler looks up or retrieves the corresponding Service Logic instance that maps to NB service request (service/network yang)
CI-2	API Handler calls Service Control Processing to perform the Service Logic on the target service or network
CI-3	Prior to CI-2, API Handler might query the (in-memory) Operational/Config Trees for the network or service details (if already existing)
CI-4	Service Control Processing retrieves the Service Logic, Config Templates, Engineering rules, and Policies as part of processing the requested action
CI-5	Service Control Processing queries and/or updates Operational/Config Trees as part of making changes to the network (VNFs/PNFs)
CI-6	Service Control Processing requests adapter layer to update/configure VNF/PNF update using the appropriate adapter for the VNF/PNF
CI-7	Service Control Processing updates the local Assigned Resources Store/Inventory once network updates are made successfully

Controller Personas Based on CCSDK Libraries

CCSDK Libraries



Controller Personas Examples
(created from CCSDK)

SDN-C Persona

NB API — Controller APIs

Control Functions

Library: Assign, Network Config, VNF Configure, Svc Function Chain, …

ODL/ Service Control

SB API — Netconf/Yang, µServices APIs, OSS APIs, Other Adapters, …

Mobility NF Controller Persona

NB API — Controller APIs

Configure & LCM Functions

Library: Rebuild, Audit, VNF Configure, Stop/Start, Scale, Svc Function Chain, HealthCk, Heal, …

Service Logic Engine, …, Other engines

SB API — Ansible, OSS APIs, Netconf/Yang, 3rd Party SFC, Other Adapters, µServices APIs, 3rd VNFM/EMS, …

Wireline NF Controller Persona

NB API — Controller APIs

Configure & LCM Functions

Library: Rebuild, Audit, VNF Configure, Stop/Start, Scale, Svc Function Chain, HealthCk, Heal, …

Service Logic Engine, …, Other engines

SB API — Ansible, OSS APIs, Netconf/Yang, 3rd Party SFC, Other Adapters, µServices APIs, 3rd VNFM/EMS, …