ONAP Modularization

Goal and Agenda

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

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

Agenda

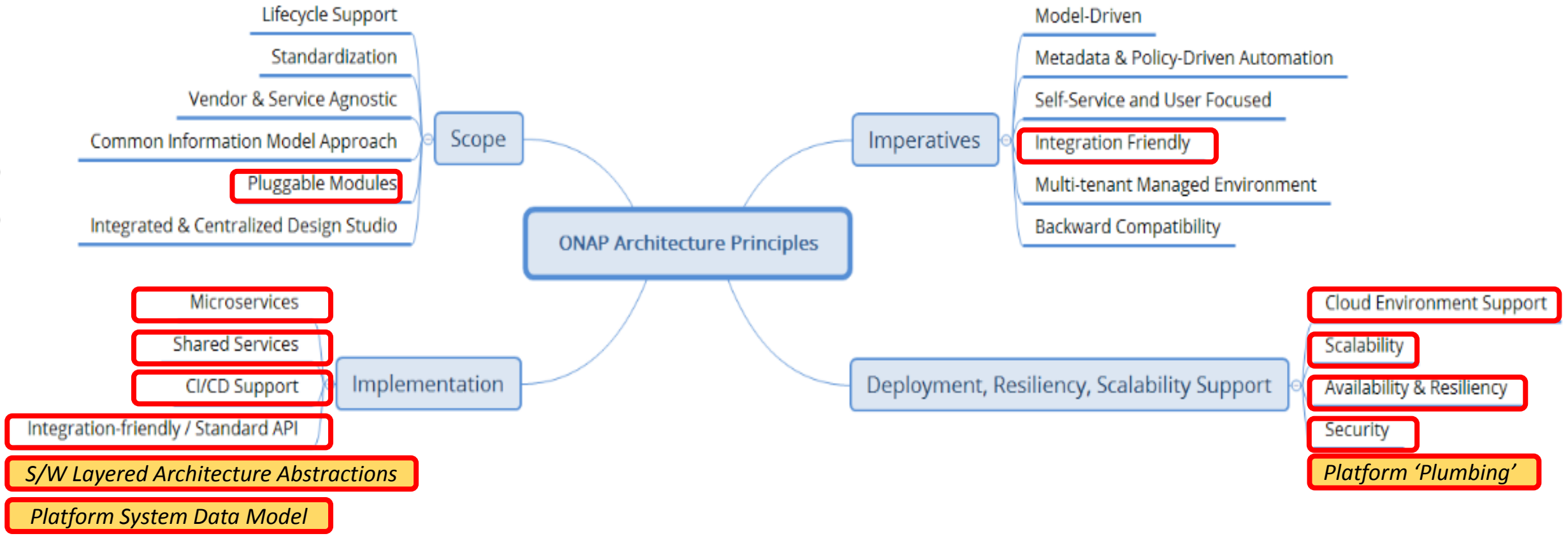
- Key ONAP challenges and critical gaps
  - Issues identified by the broader ONAP user community
- Architecture principles and approaches
  - Guide how to address the challenges
- Articulate succinct definitions and applicability of microservices, cloud-native, service mesh
- Refactor ONAP by leveraging 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

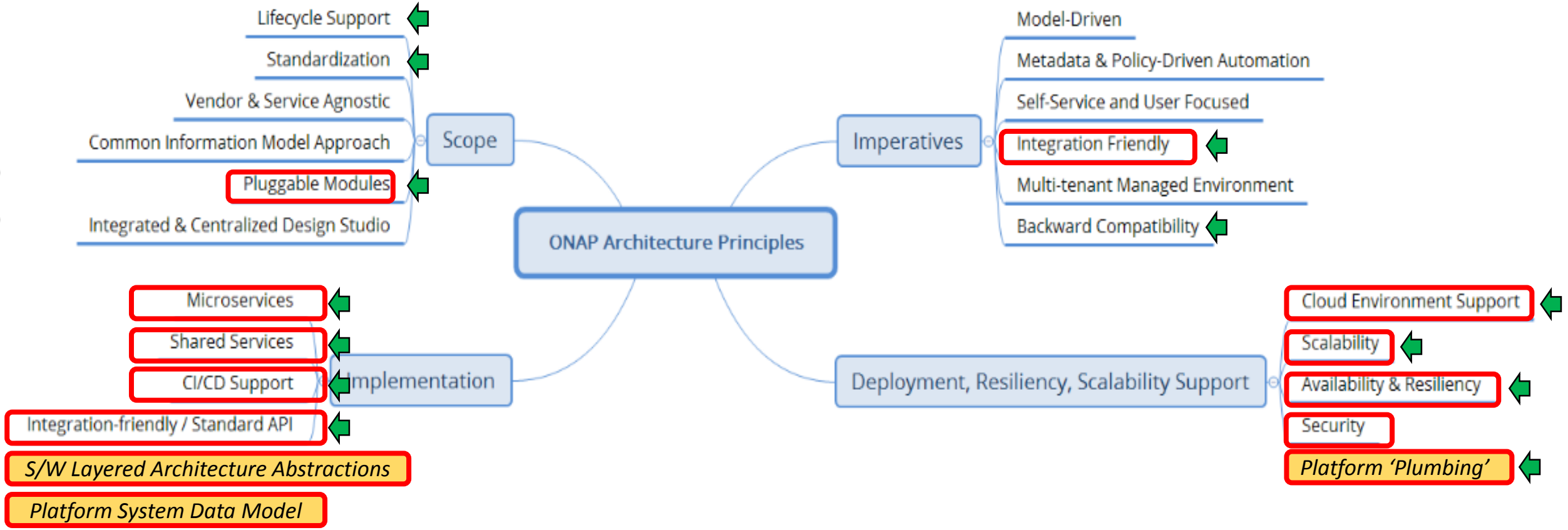
- There is a general perception that ONAP is too complex, too big and hard to make changes.
- Modules are monolithic (SDN-C, SO) and large, not sharing common utilities
- Service providers might have a specific module already implement and would like to integrate that module into ONAP (e.g. leveraging 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



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

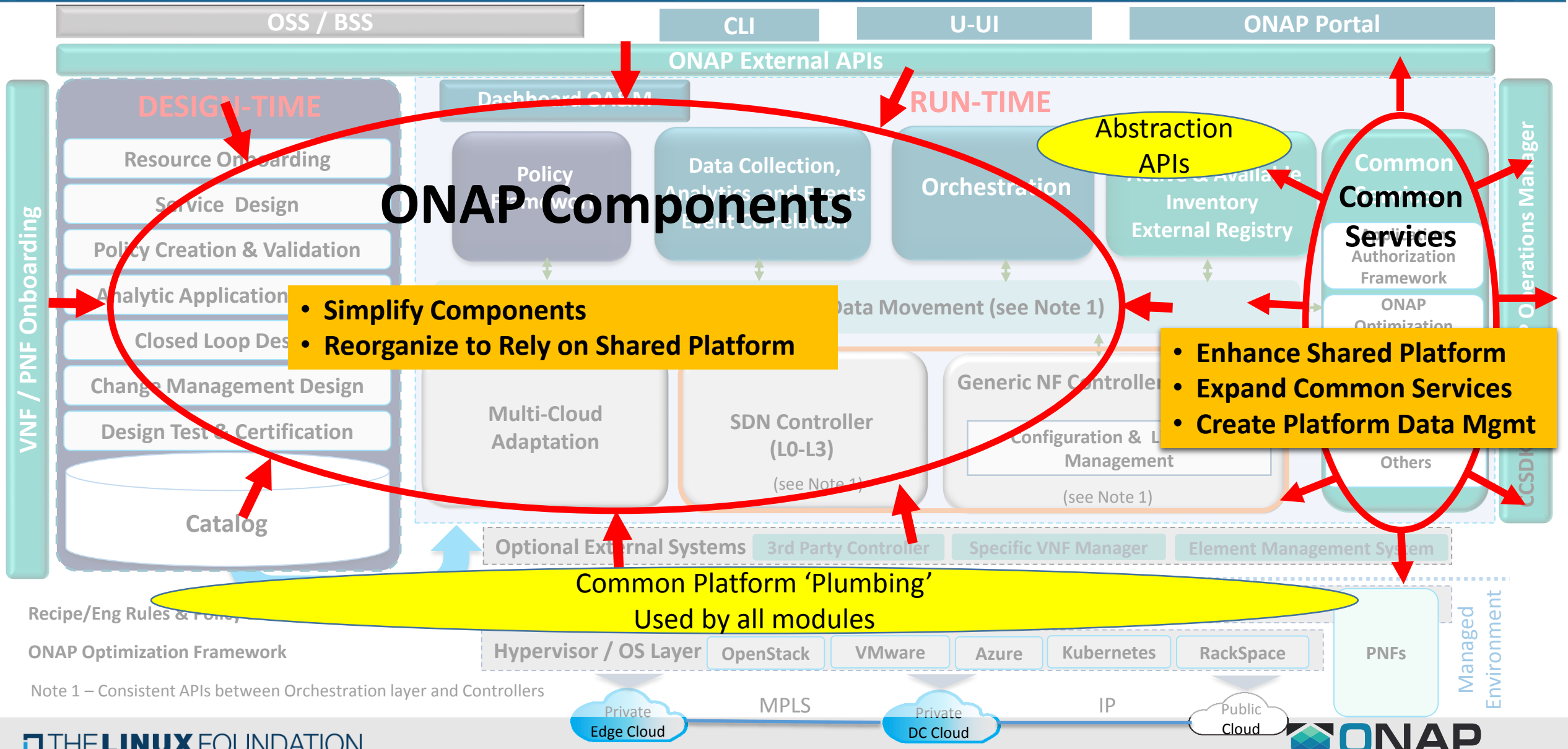
Toolings and Technologies

1. Microservices
2. Cloud-Native
   - Docker
   - Kubernetes
3. Service Meshes

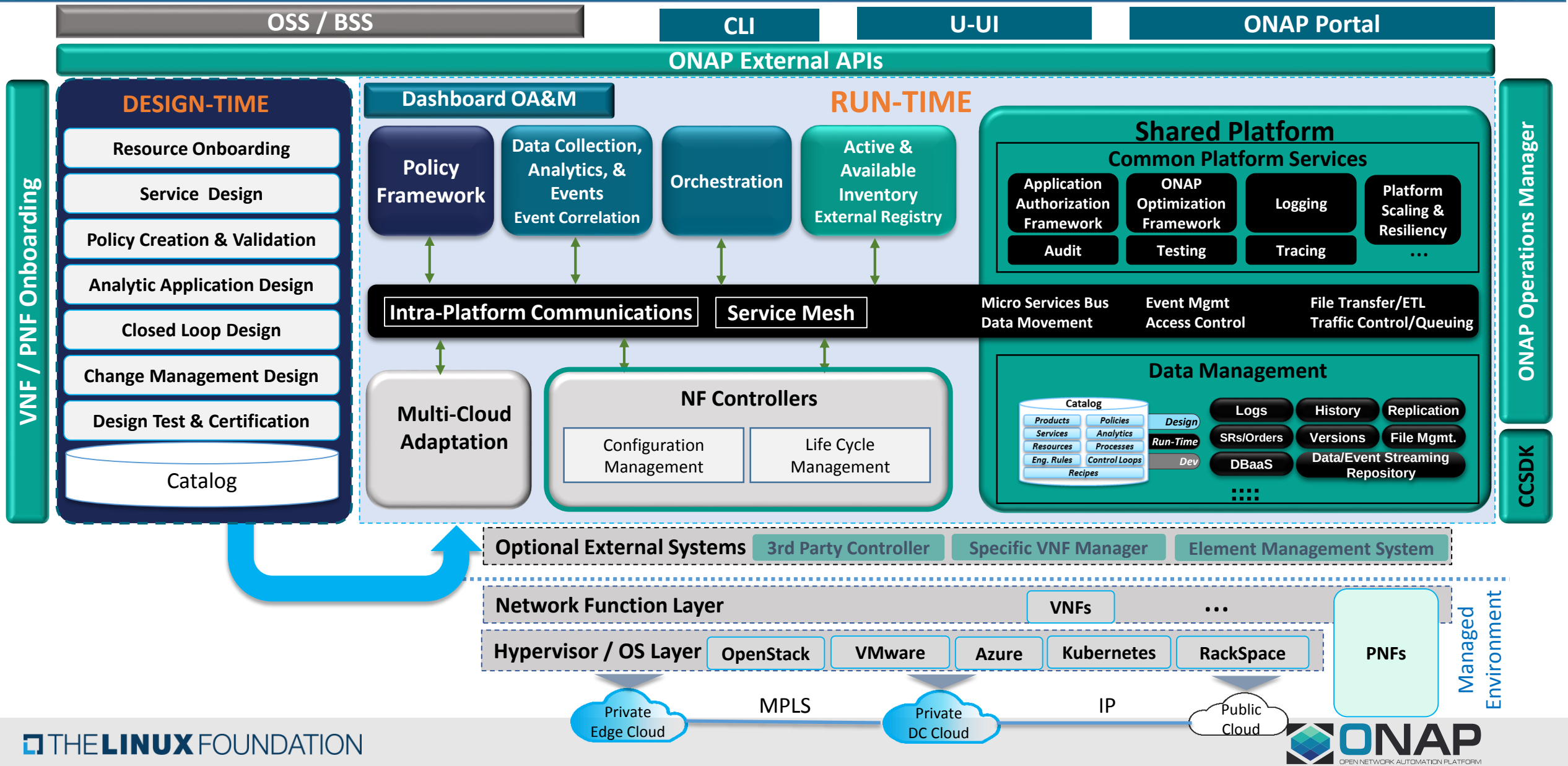
Contribute to General OSS efforts

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

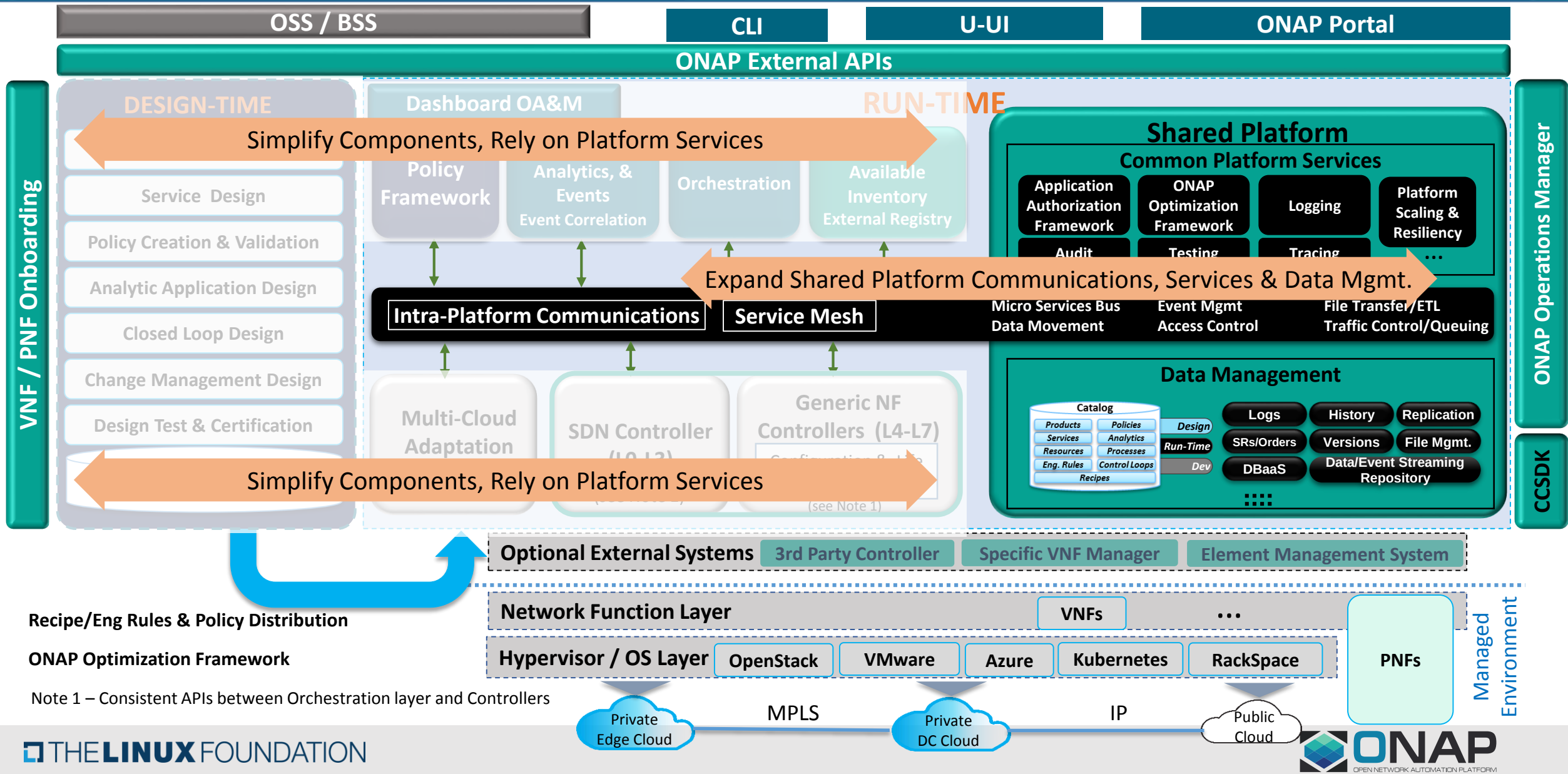
ONAP Architecture – Emphasis on Shared Platform Capabilities



Proposed ONAP Architecture Updates



ONAP Architecture – Emphasis on Shared Platform Capabilities

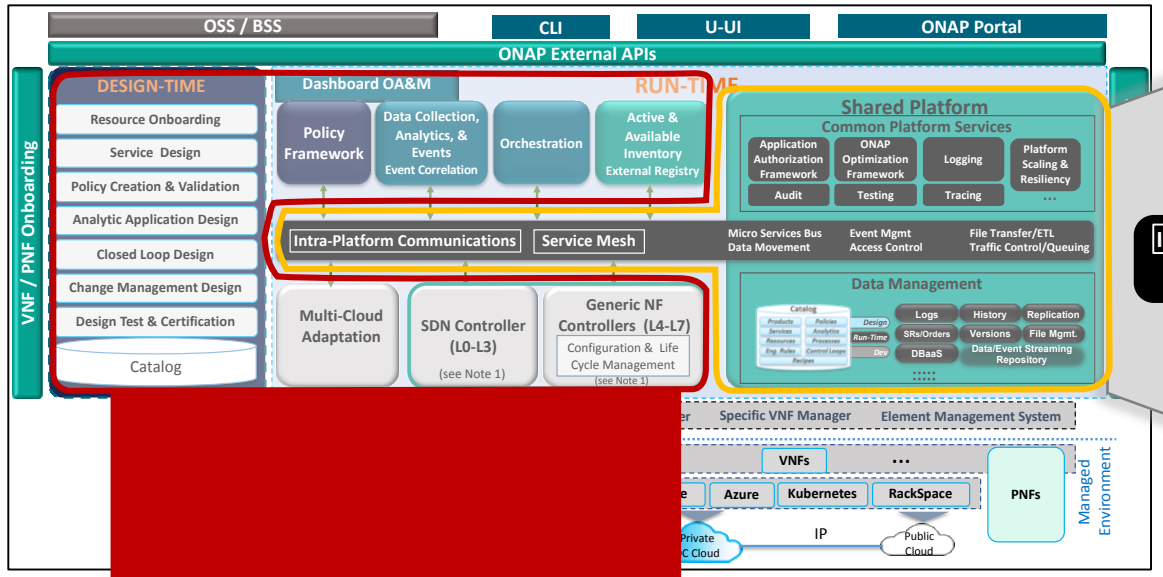


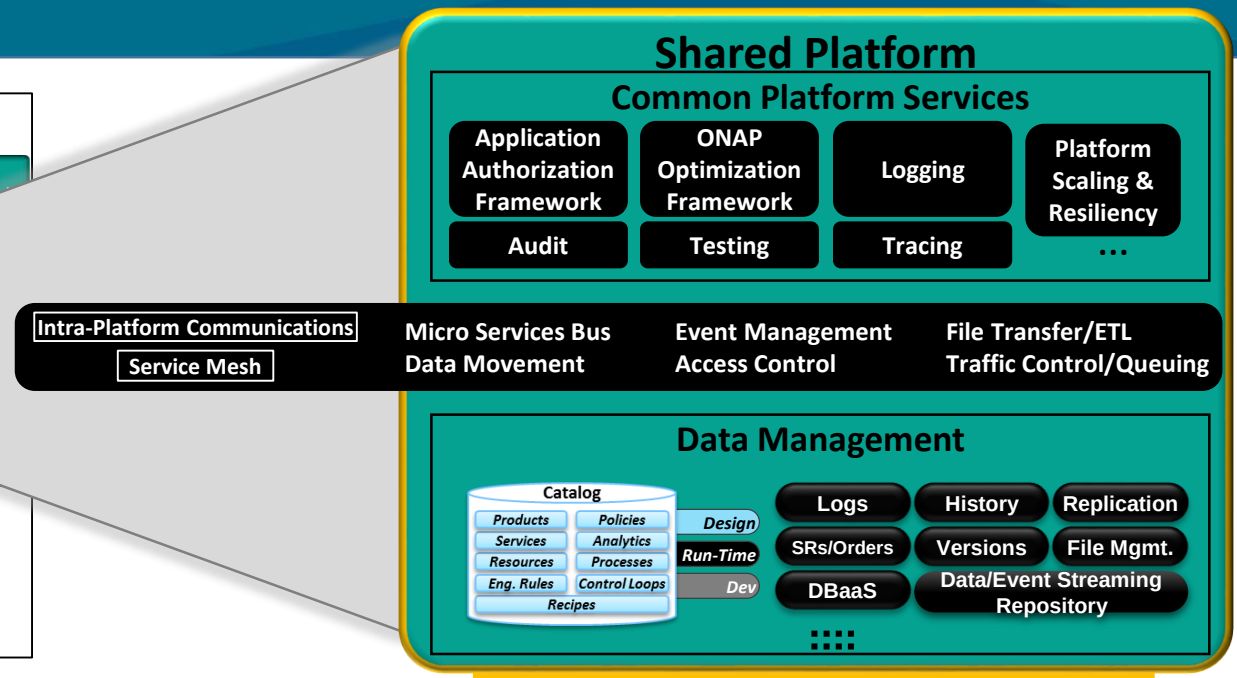
Recipe/Eng Rules & Policy Distribution

ONAP Optimization Framework

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 into smaller reusable components

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

Enabling technology swap-out

- Define abstract interfaces between components
- 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

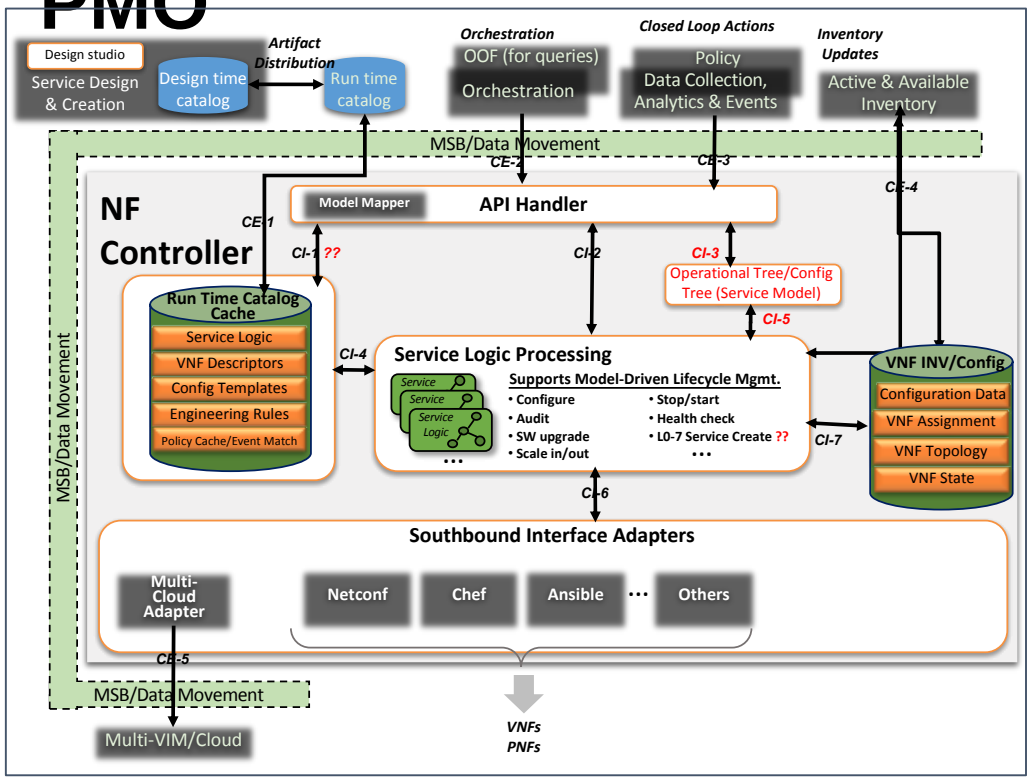
Example Component Modularization

Controllers: Current Issues and Challenges

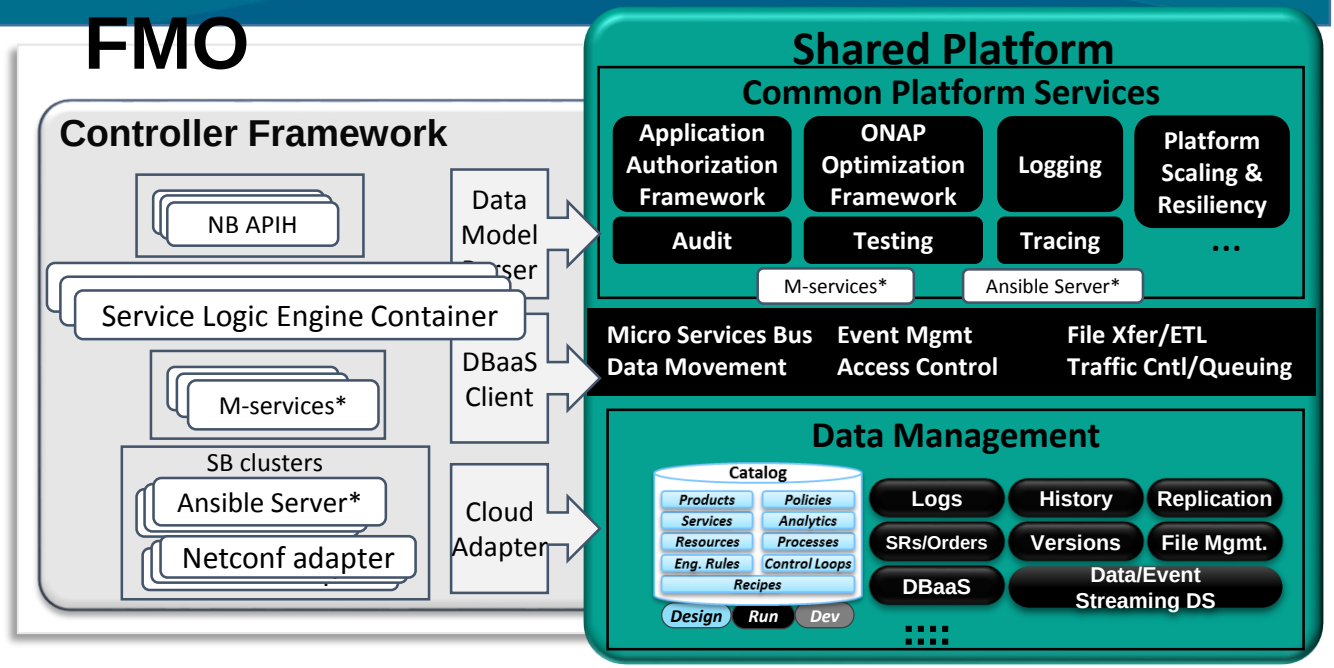
- Lack of clarity & roles in the controllers (which controller does what?)
- Divergence of controller implementation
- Duplicate and uncoordinated interfaces
  - Lack of full Configuration & Lifecycle Management by one controller
  - Lack of uniform common services in southbound interfaces
- Tightly 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