# OpenDaylight Evolution

# Whitepaper

## June 2022 v1.00

High Scale, Performance, Modularity & Open APIs

**Authored by:** PANTHEON.tech (info@pantheon.tech)

**PANTHEON**.tech s.r.o./Mlynské Nivy 56/821 05 Bratislava/Slovakia
+421 2 206 65 111/reception@pantheon.tech /www.pantheon.tech

## Introduction

OpenDaylight is at a pivot point and this paper serves to outline a proposal, on how OpenDaylight could be evolved to meet the needs of new use cases, which demand scale and performance which is out of reach for OpenDaylight (ODL) in its current guise. This white paper is to be circulated around the community to garner input and feedback to assess what if any steps need to be taken and how best to take those.

## Current Architecture

ODL is an open-source SDN controller platform, widely used in both service provider and enterprise environments. Among its key features is its ability to manage NETCONF/YANG modeled network devices and to provide REST-based APIs to the network. However, its flexible and extensible architecture allows the platform to:

- Manage a variety of diverse network devices through a variety of additional South Bound Interfaces e.g. vendor-specific CLI implementations or different management protocols (e.g. gNMI) .
- Serve as a platform for developing a variety of open source and commercial network applications (e.g. network transaction engines or BGP/PCEP protocol implementations).

The architecture of ODL is modular and layered. It is shown in Figure 1:

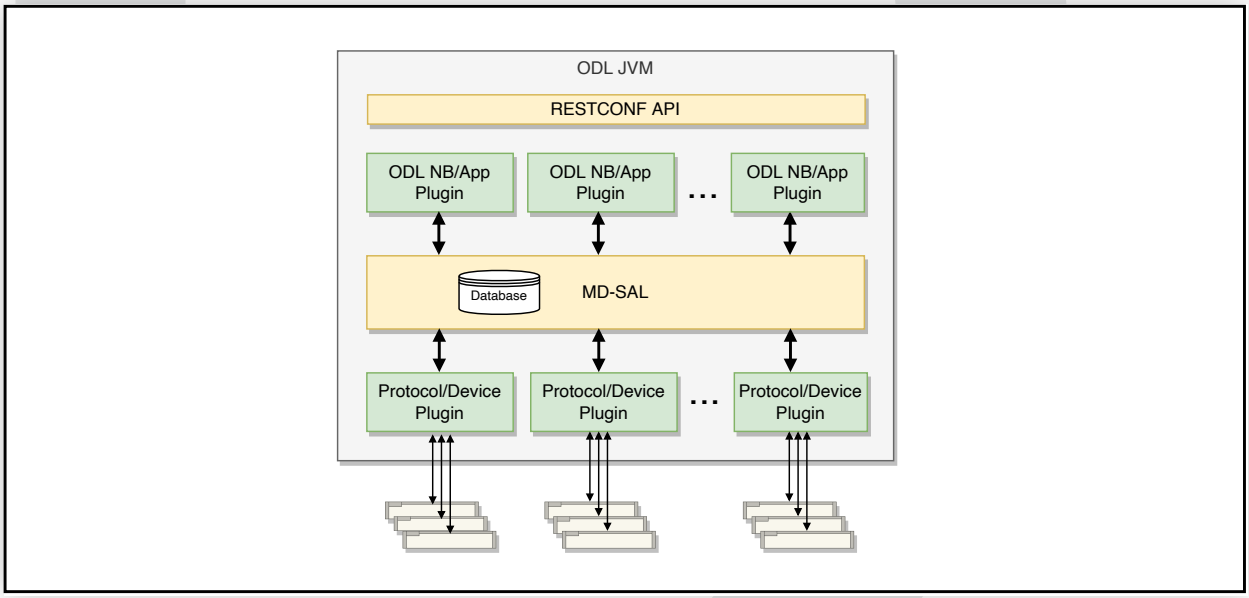


**Figure 1: OpenDaylight present mode of operation (PMO)**

The components in the above architecture are as follows:

- **Northbound/Application plugins** provide application functionality, such as Inventory and Topology managers, the BGP protocol implementation, network configurators, etc.
- **Southbound Protocol/Device plugins** provide implementations of protocols for communications with network devices, such as a NETCONF client, vendor specific CLIs or PCEP.
- **The Model Driven Service Abstraction Layer (MD-SAL)** is the “glue” that holds all plugins together and allows them to communicate with each other. It provides for all

plugins a uniform yang namespace (yang tree) and implements both a YANG-oriented Data Store where plugins store data they wish to share with other plugins and a message router that delivers messages between plugins. The uniform YANG name space seamlessly covers both network devices and objects (Normalized Nodes) in the Data Store, and so allows for elegant implementations of Topology and Inventory applications

- **Runtime** - Performance is achieved through the tight coupling of ODL components in a single address space of a JVM. With well-defined layers, components/modules, and APIs, the implementation leverages the single JVM - for function calls, efficient messaging and shared memory. However, this tight coupling also limits the ability to break out and scale individual functions/components and add value add applications
- **Key values** - the YANG data store and YANG database provide a uniform name space for both network devices and application data that needs to be shared between apps. Apps today are very specific and are in the form of “plugins” - i.e. java modules that need to be incorporated into ODL’s modular architecture and its monolithic implementation.

## Motivation

The primary motivations for evolving ODL are:

- To achieve an order of scale and performance to meet the demands of new use cases, network diversity and size (see use cases further down). This will require the scale up of the number of managed/controlled network devices, to 100s of thousands; while at the same time ensuring the Data Store is also scaled up, to hold potentially millions of objects
- Greater DevOps support for both controller core and application development enabling further enrichment of the architecture and functionality.
- Increase the applicability and attractiveness of ODL to drive greater numbers to be involved and active in the community to underpin its longevity and future in the near to longer term. Moreover, ODL needs to support a growing variety of network device types (Cisco Systems, Juniper Networks, …) and new key southbound device protocols e.g. NETCONF/RESTCONF, gNMI/gNOI, and models e.g. OpenConfig. It has to support open APIs on the Northbound and within ODL between its core components.
- The DevOps app development model implies a “cloudified” platform that can run in Kubernetes (K8s) environments in any public or private cloud. The DevOps development model for both the platform and apps that run on top of it, should support/foster better community involvement.
- Finally, the key value-add of ODL must be preserved to ensure there is no operational impact on the current ODL production deployments; moreover, features and any evolution complements and sustains the overall ODL open source project.

## Proposed ODL Evolution

The proposed ODL Evolution is an adaptation of ODL's modular and layered architecture to a cloud/DevOps environment. We first list architectural principles that guide the ODL evolution.

We then describe the new design and show how the architectural principles are reflected in design as a whole and in its individual ODL components.

Finally, we show how ODL components can be deployed in a Kubernetes cluster.

## Architectural Principles

The architectural principles that need to be encompassed are:

- **Cloud-Native Design**: In context of systems designed for deployment in K8s clusters, first and foremost all components should be packaged as K8s pods. All components should be designed as microservices that can dynamically auto-scale using cluster deployment charts & load balancing apps. A cloud-native design can leverage a variety tools and applications provided by K8s and its ecosystem, e.g. health monitoring, telemetry, deployment, upgrades, etc. For example, a deployment in a K8s cluster will facilitate updates using K8s built-in upgrade mechanisms.
- **Disaggregation of Functions into Microservices**: Disaggregation means logical separation of functions in different microservices. In context of the ODL this means that not only would different SB and NB plugins be designed as individual microservices in their own pods, but functionality within a given microservice class could be split between different microservice instances. For example, there could be different SB microservice instances for managing different device types, interface types, or network domain/areas.
- **High Availability** (HA) can be significantly improved by moving ODL to a cloud-native microservice architecture. In general, most HA solutions will have to be designed together with network devices connected to the controller and applications that will run on top of the platform. For example, consider an app design as active/active versus active/standby, or migration of legacy ODL devices and systems to the new platform, or design of inter-cluster / geo-redundant failovers in presence of multiple ODL cluster in possibly different geographical locations.
- **Tradeoff between Stateful and Stateless** depends on multiple factors, for example on scaling requirements, or whether a given transactions involves a single device vs. multiple devices, or data is persisted. For example, in southbound plugins that manage network devices, device sessions can be persistent or on-demand. If network device sessions are persistent, then a stateful config load balancer may be required to assign sessions to different plugins; if on the other hand, network device sessions are on-demand then a regular stateless K8s load balancer can select a lightly loaded plugin instance which then sets up the session to the device.

Other considerations for determining whether an app is stateful vs. stateless are based on an understanding of other functionality that exists or maybe required as dependencies at scale demand more automated approaches to operations. For example, service intent validations, network-wide transactions, or snapshots/rollback are most often stateful, while per-device can be stateless.

- It is recommended to go stateless wherever possible. Stateless microservices are easier to scale (just start another instance) and easier for HA (simple restart/rebalancing will do as opposed to having to have an always costly stateful redundancy scheme). However, state needs to be kept somewhere. Hence, the model where you can keep the business logic in microservices stateless and keep their respective state in an external Data Store (it could be the ODL DB cluster, a NoSQL DB cluster, or an SQL DB) may be the best.
- **Use modeling to deal with the diversity of devices** is key to handle the diversity of devices at scale. It allows to create automating techniques that normalize diversity of devices with inherent nuances on interfaces, software version, device capabilities, etc, Data schemas for all network devices managed by the controller can be either translated to a common device representation, such as an OpenConfig model, or passed transparently to the northbound apps. Choice of modeling language and translation also plays a role: there is YANG for devices, while NB APIs are better defined in terms of OpenAPI/JSON/protobufs. Tooling is required to automatically translate between YANG schemas and protobufs/JSON. There could be dedicated “translation microservices” that translate between formats/models and/or dedicated “translation microservices” for mapping of different device capabilities.
- **Security**: The evolved ODL platform and any solution based on it will have to comply with industry standards with respect to security controls, access, separation and hardening. The access to devices from the controller and vice-versa must be secure. The ODL cluster itself must be secured, especially if it is geo-distributed. The NB APIs and microservices within the cluster must be secured using proven K8s tools such as RBAC access control, SPIFFE / SPIRE. For example, only communication between microservices that need to talk to each other will be allowed.
- **Scale Up/Down**: In order to define the scaling requirements for a system, the cadence of transactions per device needs to be understood. This is not limited to just configuration, but one needs to look at the overall number of operation per second  per device, and consider the maximum load during inevitable bursts. In a K8s environment adaptation to changing load can be achieved by dynamically adding/removing microservices to/from the cluster.

- **Ease of operations**: Cluster and microservice monitoring can be provided by K8s. In addition, there can be an ODL specific “Cluster Monitor / Cluster Operator”. For debugging, logs provided by K8s and a logging microservice (for example, Logstash) can be used. For call flow logs, distributed tracing projects can be used, such as *jaegertracing.io*, *zipkin.io*, or *opentelemetry.io*.

## Proposed Architecture Overview

The proposed architecture is a cloud-native reinterpretation of the original ODL architecture. It’s both modular and layered, similar to the original ODL. The layers are roughly the same as in the original ODL - northbound, southbound and a glue in the middle that binds them together. However, components that comprise the ODL system are now K8s pod based microservices rather than Java modules.

The proposed high-level architecture of the evolved ODL is shown in Figure 2.

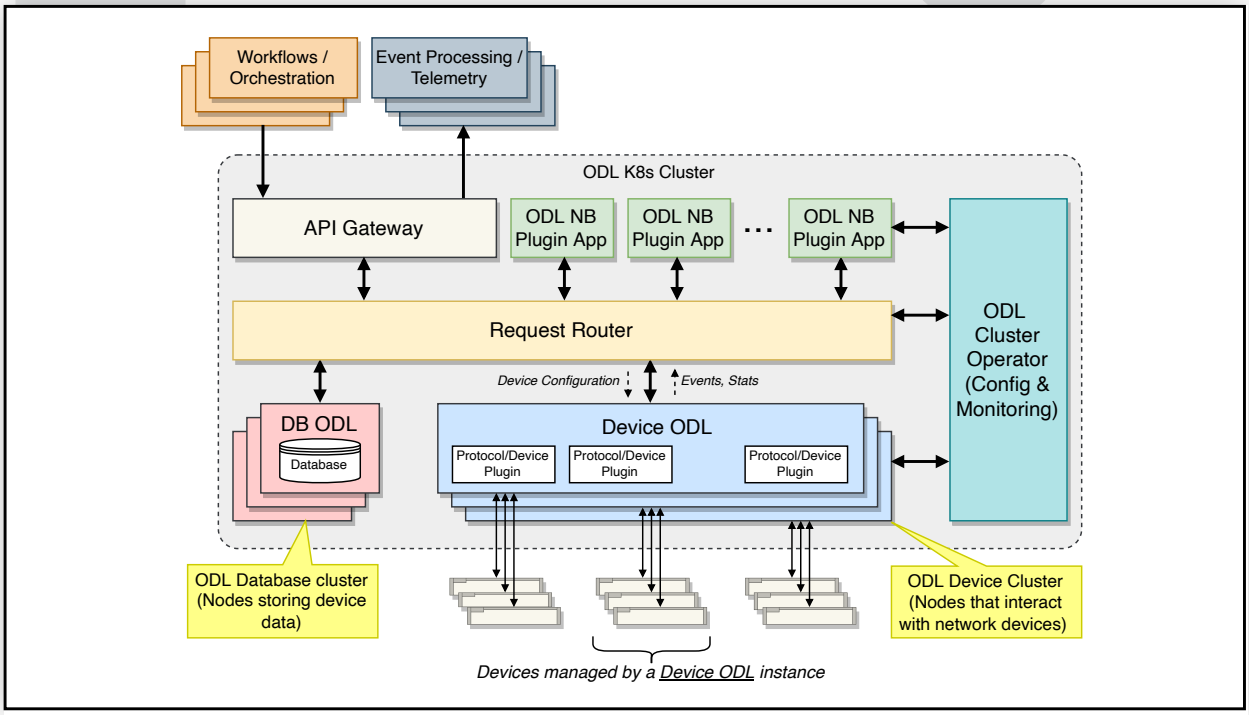


**Figure 2: Architecture of the evolved ODL**

The components in the architecture are as follows:

- **Device ODL**: a distributed, linearly scalable interface to network devices. Each Device ODL instance can manage a certain number of devices (given mostly by the number of active

sessions that the underlying OS can maintain), and scaling to 100s of thousands of devices is achieved through adding Device ODL instances.

- Support for both persistent and on-demand sessions to network devices can be provided; According to the Disaggregation principle, different Device ODL instances can be tailored to different network device types.
- **Northbound (NB) Apps/Plugins**: Container based (K8s pod based) re-implementations of existing ODL NB plugins, such as BGP/PCEP, and new K8s pod-based apps that can extend the ODL functionality, such as a model translator, a workflow engine or a network transaction engine.

  NB apps/plugins become microservices - they become more like standalone apps with their own implementation (no longer limited to Java) and operating environment. .
- **Request Router (RR)**: In a distributed system, messaging replaces monolithic system's function calls and shared memory as the fundamental communication system.

  Sure, there must be optimizations, such as prefetching and caching, but the very foundation is messaging. The key to acceptable overall system performance is a low latency and high bandwidth messaging system. In a distributed ODL such messaging system has an additional core function - it provides seamless logical view of the yang name space (the tree) to its clients.

  In other words, it needs to keep track of where Normalized Nodes are physically located, and when an app wants to access a Normalized Node, it routes the app's request to its location.
- **API Gateway**: Provides RBAC security approach for controlled access to platform APIs and to APIs of app microservices that are deployed in the ODL cluster. The API Gateway is the equivalent of the Northbound API in the current ODL.

  It is stateless, so it can be easily scaled up/down by adding/removing instances.
- **Security:** the implementation of security measures zero trust, only allow communication between trusted/authenticated/authorized clients.
- **DB ODL**: replicated and highly available YANG database, reusing as much as possible from the existing ODL Data Store; Scaling is achieved through partitioning of the overall YANG tree into subtrees and letting individual DB ODL instances manage the subtrees.

As for HA, each DB ODL instance could be made redundant, either active-standby or active-active. A discussion within the ODL community is needed about consistency models if/when replication is utilised (e.g. strict vs. eventual consistency).

For DB ODL microservices that need to keep metadata for device schemas this metadata could be loaded on as-needed basis, i.e. only into nodes that store particular subtrees or type of data.

Moreover, some DB ODL instances could be augmented to provide Value Add services, such as Active Inventory. The number of ODL nodes in the system depends on the overall number of Normalized Nodes that need to be stored in the controller.

In other words, the number of DB ODL instances can grow with the overall number of Normalized Nodes that need to be stored for all apps using the DB ODL system.

- **ODL Cluster Operator**: Provides system configuration and monitors the health of the ODL cluster. The implementation will be based on the K8s Operator Framework.

---

The biggest challenge is to create a high performance distributed equivalent of the original MD-SAL that can maintain a uniform YANG name space in a highly distributed system.

DB ODL, together with the Request Router, provide the uniform YANG name space to SB and NB apps that spans objects in both network devices and in the Data Store. As such, together they provide functionality similar to the original MD-SAL.

## Managing the YANG Name Space

YANG name space is a tree. An example of a YANG name space (tree) is shown in Figure 3:

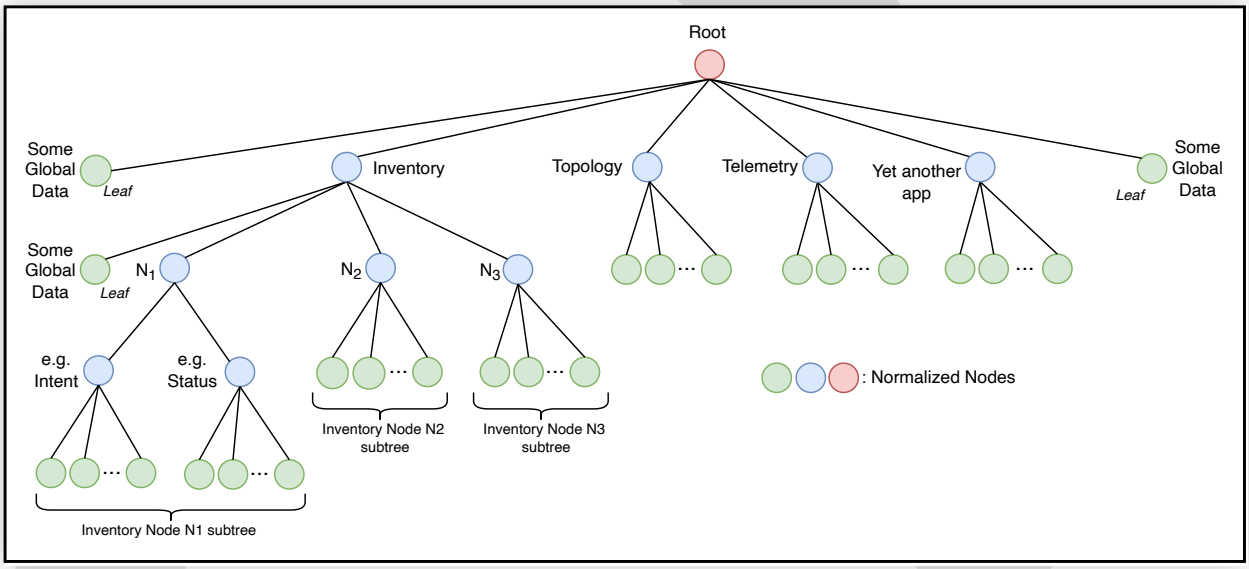


**Figure 3: YANG Name Space**

The example name space (tree) has a few subtrees, such as Inventory, Topology, Telemetry, or some other app subtree. It can also have some global data, as shown by a couple of Leaf nodes hanging off the Root node.

Subtrees may have sub-sub-trees, as can be seen with the Inventory subtree, where sub-sub-trees correspond to network devices that connect to the controller and their yang data stores are mounted into the controller's name space.

A subtree may have some subtree-global data that applies or is common to all its sub-subtrees. While there is no theoretical limit to the number of subtree levels, in practice it (and the number of Normalized Nodes in the tree) will be limited by applications' needs.

The subtrees are managed by different applications, as shown in Figure 4:

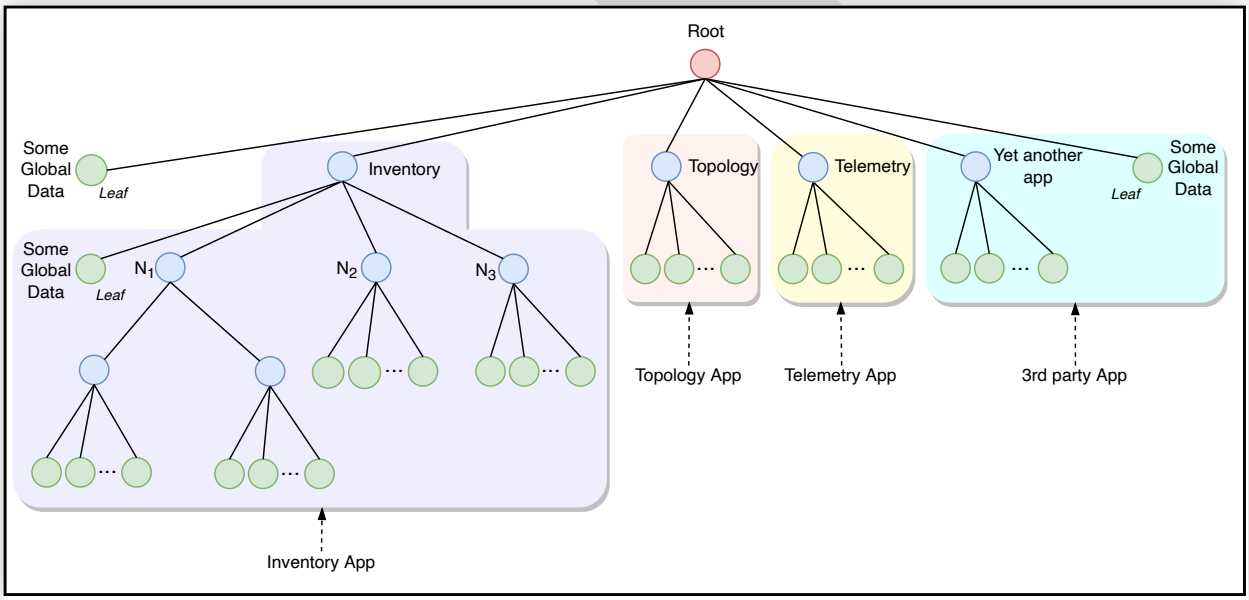


**Figure 4: Logical ownership of data in the YANG tree**

The figure shows examples of applications that logically "own" subtrees (i.e. portions of the overall YANG name space), such as the Inventory app, the Topology app or the Telemetry app. The YANG name space is public, and typically contains data that apps want to share with each other. Apps can also have private data stored in other data stores of their choice (e.g. in local memory, on a file system or externally in a database). Also, apps (or admin) could set different access rights for other apps to access their subtrees.

The key to scaling the number of elements in the YANG name space (the tree), is to partition it into subtrees and let different entities "physically" own the subtrees. The problem is that the physical ownership of data does not overlap 1:1 with the logical ownership of data. Data in the tree can physically reside in the Data Store or in mounted network devices. To scale the Data Store, it has to be sharded, i.e. there must be multiple DB ODL instances and different must subtrees reside in different DB ODL instances. To scale the number of network elements that the controller can manage at the same time, there must be multiple Device ODL instances, each managing a certain number of devices. This is all shown in Figure 5:

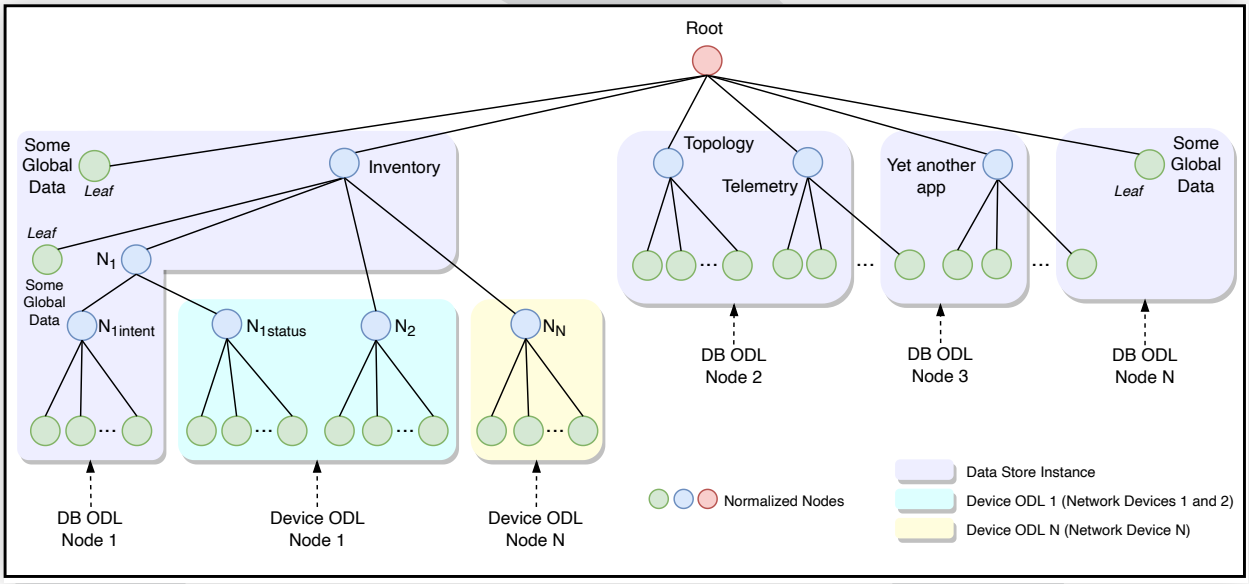


**Figure 5: Physical location of data in the yang tree**

The sharing needs to factor performance to ensure the correct distribution of data etc. coupled with Replication, caching in MB clients, local, pre-fetching, consistency models. In the figure, we see elements in our example tree residing in one of N Data store shards or in one of N network devices. Note that for Data Store objects, the shard boundary can run at any subtree level: for example, the entire Topology tree resides in DB ODL Node 2, whereas the Telemetry tree is split between DB ODL Nodes 2 and 3. The shard boundaries should be made invisible to apps and dynamically rebalanced by the system to achieve best performance and scale. Eventually, the number of DB ODL nodes could be dynamically scaled up and down based on the application needs.

In Figure 5, we also see that some applications may read/write/create objects in both Data Store and Network Device subtrees. For example, an Inventory application may store the configuration intent data in the data store, and the real config data in the Network Element.

Of course, the operational data would also be read from the Network Element. Figure 5 shows a couple of Network Elements managed by Device ODL 1, and another Network Element managed by Device ODL instance N.

When an app wants to read/write/create an object at a certain position in the tree, the system must first find the Data Store shard and DB ODL instance or the Network Element and the Device ODL instance that is responsible for the object. This is done transparently to the application by the Request Router described in detail in the next section.

## The Request Router

The Request Router (as its name suggests) routes requests to read/write/create objects (Normalized Nodes) in the yang name space (tree) to those entities that physically hold (them or should hold them in case of create requests). In addition to read/write/create

Normalized Nodes, apps can register to be notified when a change is made to a given subtree. Another role of the Request Router is to keep track of these registrations and to route notifications to the registered app. Also, the Request Router must deliver yang Notifications from devices that generate them to apps that register for them.

To achieve acceptable system performance, the Request Router must provide low latency and high bandwidth. Therefore, clients/apps connected the Request Router must communicate directly with each other (point-to-point), which in turn means that the Request Router's data plane must be distributed, where a data plane instance is co-located with each client that uses the Request Router.

In other words, the Request Router must contain a Zero-MQ like message bus implementation, which is shown in Figure 6:

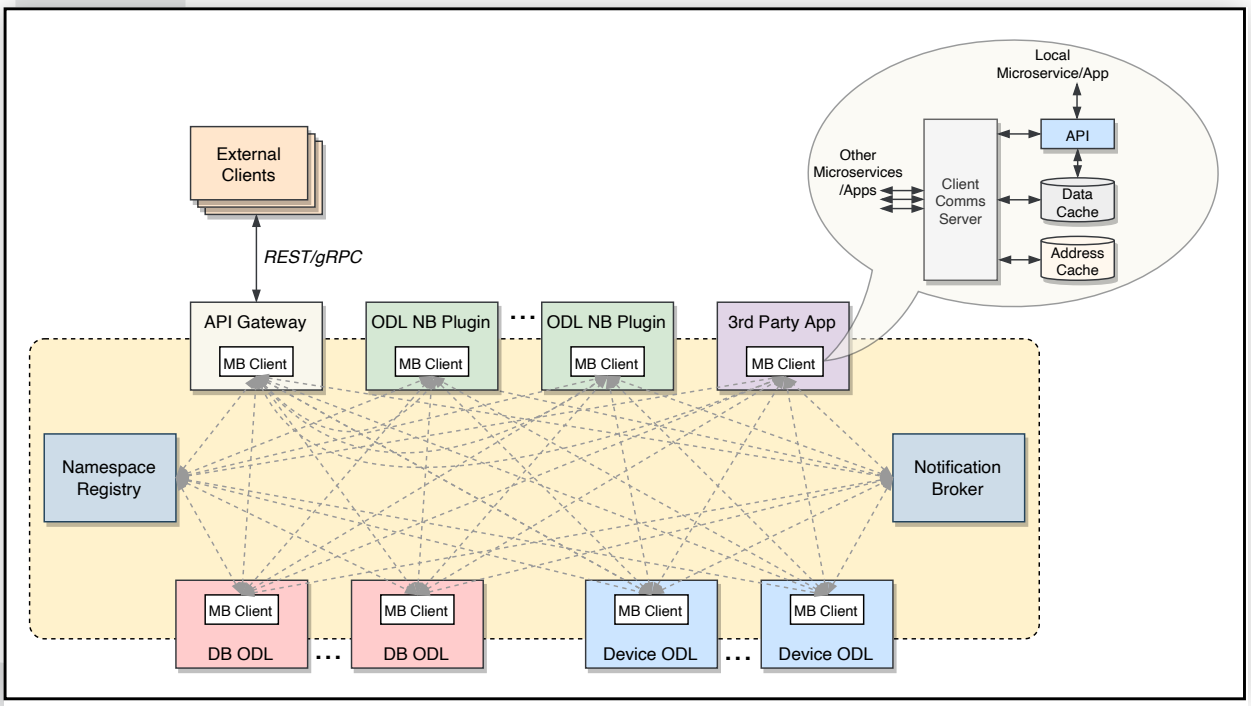


**Figure 6: Request Router details**

The Request Router components are as follows:

- **Message Bus (MB) Client**: embedded with each application, most likely as a sidecar container in a pod deployment. Provides the equivalent of today's MD-SAL API to apps and communicates with other MB Clients in other applications. When an MB Client's app registers via the API for YANG notifications from network devices, the MB Client registers with the Notification Broker and passes the notifications from the Notification Broker to the app. When an app wants to read/write/create a Normalized Node in the YANG name space, the MB Client gets the information about the Normalized Node's physical location from the Namespace Registry and then direct the apps' request to the entity that holds the object. For entities that physically hold Normalized Nodes (DB ODL and Device ODL), the MB Clients also register their portions of the YANG name space in the Namespace Registry.

  Security could be built into the client, or just deployed around the client (and/or the app) in the K8s cluster. For example, SPIFFE/SPIRE would be used to secure the communication between MB Clients. SPIRE may be deployed independently of the client, in another sidecar.

  The MB Client further consists of:

  - **Client Communications Server**: does the heavy lifting of communicating with RR infra components (Name Registry, Notification broker) and other MB Clients
  - **API:** interface to apps, the Request Router part that is visible to applications. Must be compatible with the existing MD-SAL API and additionally provide functions that allow for development of high-performance apps. Both RPC (synchronous and asynchronous) and pub-sub APIs must be provided to apps.
  - **Data Cache**: holds cached and prefetched data; Apps can iterate on data located in the cache.
  - **Address Cache**: When the MB Client wants to access a remote Normalized Node for the first time, it does not know its location and must query the Namespace Registry. The MB Client remembers the Normalized Node to location mapping in its Address Cache for subsequent accesses. Alternatively, the Namespace Registry can broadcast subtree locations to all MB Clients.

- **Namespace Registry**: Maintains information about the physical location of Normalized Nodes in the YANG namespace (tree) and responds to request for such information from MB Clients.
- **Notification Broker**: Manages registrations for yang Notifications that come from network devices. When a notification is received from one of the Device ODLs, it fans it out to all registered MB clients.

We need to figure out consistency models and the cache invalidation / cache timeout times. Most likely eventual consistency might be the best consistency model, as opposed to strict consistency that is used in today's ODL.

**Other aspects** to consider in the Request Router design are:

- The need for efficient wire encoding. For security reasons, traffic on the wire may have to be encrypted in most deployments.
- Local caching of data in MB Clients
- "Address Server" and local caching of Normalized Node locations in MB Clients
- Address space is probably not aggregatable (maybe in the DS, but not for devices), which has implications on the scale & performance of the Namespace Registry and the size of the MB Client's Address Cache.
- Scale**:** need to scale Namespace Registry and Notification Broker; This can be done by adding instances of each. Since MB Clients are distributed, they scale naturally with the number of apps (in other words, it can safely assumed that a client is never a messaging bottleneck, so if an app is overwhelmed by messaging, we need to scale up the app first long before we would have to scale the MB Client)

## Request Router Example Flow

To better explain the design of the Request Router, let's examine a flow, for example a read request flow. It is as follows:

1. Local App issues a Read
2. API goes to Comms Server to perform the Read operation
3. Comms Server looks in the local Address Cache for one or more locations where data is located (if there are multiple copies, it selects one at random)
4. If no location is found, the Comms Server queries the Namespace/Address Registry for locations and stores the query results in the Address Cache
5. The Comms Server requests data from one of the locations and puts the results into the local cache.
   - Some data may be prefetched for further iteration by the app on the data (i.e. more data may be retrieved than what was requested by Local App, depending on the DB technology).
   - The Comms Server may combine multiple subtrees coming from different locations and then put all incoming data into the Data Cache.
6. Response with the first data element returned to Local App. Local App then may iterate over data in the Data Cache.

## Deployment

The deployment of the controller in an K8s cluster is shown in Figure 7:

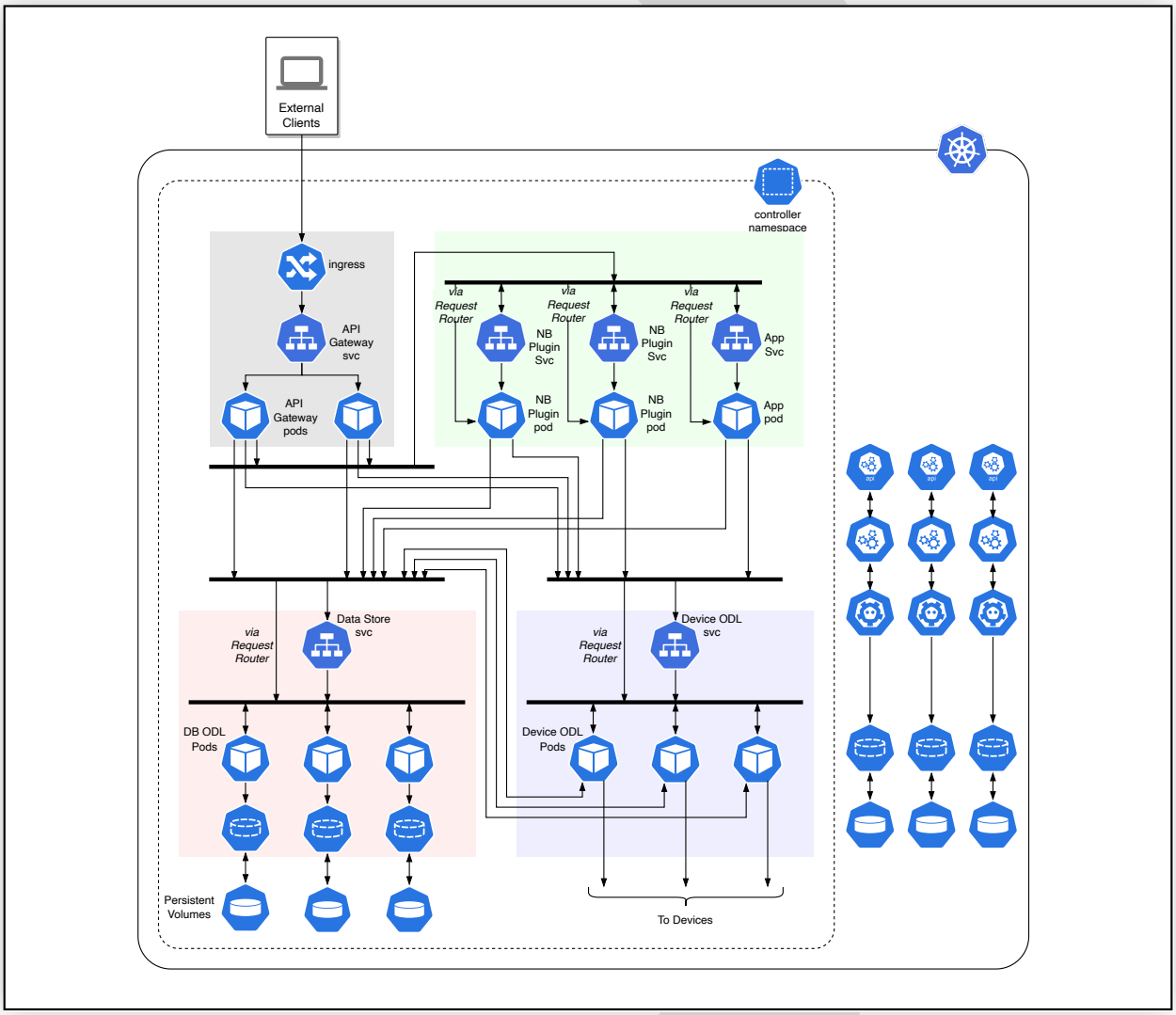


**Figure 7: Controller deployment in a Kubernetes cluster**

As can be seen from the figure, the deployment takes advantage of K8s services. Device ODL and DB ODL APIS should be accessible both via the Request Router and via their respective APIs.

Northbound / Application plugins could decide (based on performance requirements) whether they expose their API via a K8s service or through the Request Router. The figure also shows the need to manage persistent volumes within the K8s cluster.

As can be seen from the figure, the deployment takes advantage of K8s services. Device ODL and DB ODL APIS should be accessible both via the Request Router and via their respective APIs.

Northbound / Application plugins could decide (based on performance requirements) whether they expose their API via a K8s service or through the Request Router. The figure also shows the need to manage persistent volumes within the K8s cluster

- Important component: ODL Cluster Operator
- Figure shows three DB ODL instances, but the number of instances can be scaled up as needed. Will need persistent volumes to store DB ODL's data
- Device ODLs linearly scaled as needed
- MB Clients deployed in sidecar containers into ODL app and platform microservice pods
- Eventually, deployment in all clouds and different platforms, e.g. Anthos or OpenShift

# Solutions & Use Cases

This section summarizes a few example use cases which will require OpenDaylight to scale in terms of performance, resilience and extensibility.

Whether this is use case which requires 500K devices to be managed or new services to be added to augment functionality, each will drive a unique requirement from an evolved ODL.

## Transport PCE / Network Optimization

An existing project, the Path Computation Engine (PCE) computes paths on behalf of the nodes in the network for both Packet & Optical layers, selecting the required path as predicated by policy and instantiating the respective overlay service.

PCE is an application that is run on top of ODL and is an example of how an evolved ODL would simplify the applications extensibility of the controller, for example by adding new microservice-based implementations of path computation algorithms to the controller, or augmenting existing algorithms by plugging them into workflows

## Edge Computing

With the momentum of services and functions needing to be closer to customers at the Network Edge rather than the Cloud, SDN has become key to driving intelligent, fast and dynamic overlay networking paths between CPE, decentralized edge and centralized cloud.

## IoT

Complementing and leveraging Edge Computing is the requirement to support the proliferation of smart IoT devices. The volume and diversity of devices drives a need for scale and performance for:

- Reduced latency between IoT devices and processing function.
- Faster response times and increased automation for operational efficiency.
- Improved and agile network bandwidth.

## Service Orchestration/Chaining

The inclusion of workflow and decision automation applications, with tools for creating workflow and decision models.

This facilitates the chaining of configurations and components to complete network services, orchestrating multiple devices, via the same or disparate interface types.

## Edge, Fabric Controllers

Supporting hybrid architectures with basic L2, L3 data planes in hardware devices / on premise and high touch processing (NAT, FW, VPNs, etc) in CNFs, in a cloud - either in provider's DC or a cloud provider.

## RAN & 5G

The controller, which manages packet-flow control to enable intelligent networking, is situated between network devices and applications.

In this architecture, network control becomes programmable. With the controller, administrators will be able to easily manage the 5G network and introduce new services or changes.

## Hierarchical Controllers

Some present day ODL use cases feature hierarchy of a super controller controlling multiple sub-controllers, where each sub-controller in turn controls a domain.

The cloud-native architecture can do away with such approach, as both the super controller and sub-controllers can be just apps in an overall controller deployment.

## Conclusion

This white paper has been authored to provoke thoughts and feedback as to where and how OpenDaylight, if evolved could meet your demands for continued open source innovation in supporting existing and new use cases.

We would welcome your thoughts and comments on this white paper and look forward to hearing further from you.

## Glossary

- **Northbound / Southbound**: with respect to ODL APIs, *Northbound APIs* denotes APIs towards applications that use the controller, *Southbound APIs* denotes APIs towards network devices
- **Data Store**: YANG Data stores are a fundamental concept binding the data models written in the YANG data modelling language to network management protocols such as NETCONF and RESTCONF. YANG Data store architecture is defined in RFC7950
- **RPC**: Remote procedure call is when a computer program causes a procedure (subroutine) to execute in a different address space (commonly on another computer on a shared network), which is coded as if it were a normal (local) procedure call, without the programmer explicitly coding the details for the remote interaction
- **PCEP**: Protocol to communicate between a Path Computation Element (PCE) and a network device. In context of this white paper we refer to the Stateful PCEP protocol defined in RFC 8231.
- **BGP**: Border Gateway Protocol, is a standardized exterior gateway protocol designed to exchange routing and reachability information among autonomous systems (AS) on the Internet. It is defined in RFC 4271.
- **NETCONF** is an IETF network management protocol that provides an administrator or network engineer with a secure way to configure network devices. It is defined in RFC 6241.
- **YANG** is a data modelling language for the NETCONF configuration management protocol. Together, NETCONF and YANG provide the tools that network administrators need to automate configuration tasks across heterogeneous devices in a software-defined network (SDN).
- **Apps**: applications
- **gNMI:** a unified management protocol for streaming telemetry and configuration management that leverages the open source gRPC (https://www.grpc.io/) framework.
- **gNOI:** defines a set of OpenConfig gRPC-based microservices for executing operational commands on network devices.

- **DevOps:** a combination of the terms development and operations, meant to represent a collaborative or shared approach to the tasks performed by a company's application development and IT operations teams.
- **K8s**: Kubernetes is a portable, extensible, open source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation.
- **DB**: A *Database (DB)* is an organized collection of structured information, or data, typically stored electronically in a computer system. A database is usually controlled by a database management system (DBMS). Together, the data and the DBMS, along with the applications that are associated with them, are referred to as a database system, often shortened to just database.
- **SQL:** Structured Query Language, which is a computer language for storing, manipulating and retrieving data stored in a **Relational Database**. SQL is the standard language for all Relational Database Management Systems (**RDBMS**).
- **NoSQL:** an approach to database management that can accommodate a wide variety of non-relational data models, such key-value, document, columnar and graph…
- **Protobufs:** Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data – think XML, but smaller, faster, and simpler. You define how you want your data to be structured once, then you can use special generated source code to easily write and read your structured data to and from a variety of data streams and using a variety of languages.
- **SPIFFE**: Secure Production Identity Framework for Everyone, is a set of open-source standards for securely identifying software systems in dynamic and heterogeneous environments. Systems that adopt SPIFFE can easily and reliably mutually authenticate wherever they are running.
- **SPIRE**: an open-source toolchain that implements the SPIFFE specification in a wide variety of environments.
- **Logstash:** a light-weight, open-source, server-side data processing pipeline that allows you to collect data from a variety of sources, transform it on the fly, and send it to your desired destination.
- **Normalized Node:** an *OpenDaylight YangTools* API for working with YANG Data. It is defined here.