First CORD Release:

Building and Using CORD

Central Office Re-architected as a Datacenter

#OpenCORD

David Bainbridge, Ali Al-Shabibi

Building a CORD POD

#OpenCORD

Automation Concepts

Types of automation in CORD

- Deployment automation
  - Going from bare metal to POD as easily as possible)
  - Provides framework for CI/CD
- Operational automation
  - Scaling POD resources (up/down) as they are plugged/unplugged
  - Each component viewed as a Field Replaceable Unit (FRU)
- Test Automation
  - Correctness testing during deployment and operation

Server Roles

Servers will fill typically either of these roles:

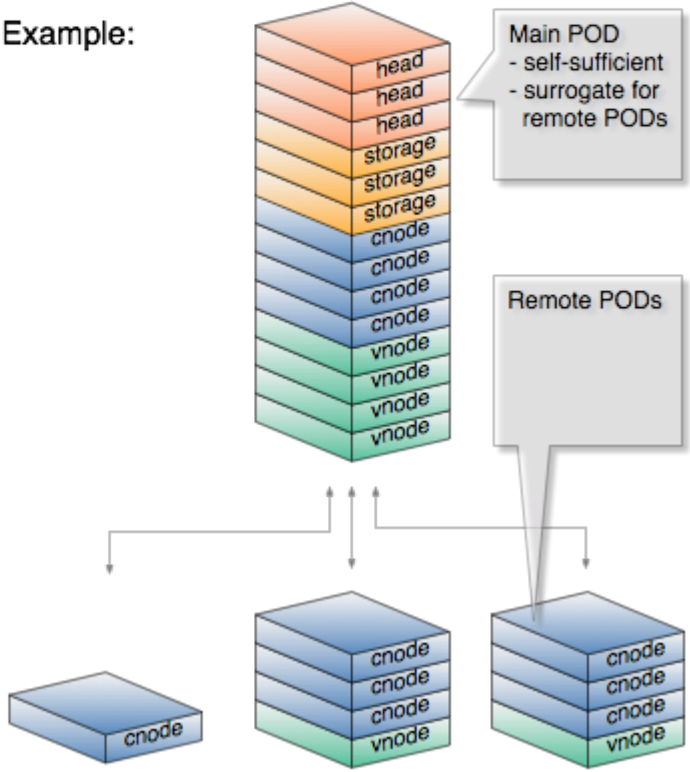
- **Head** node
- **Storage** node
- **C-node**: compute node for pure containers
- **V-node**: OpenStack compute nodes (for VMs)

Hybrid nodes may be supported too

Smaller PODs (micro and mini) may be headless: controlled from a remote (shared) head

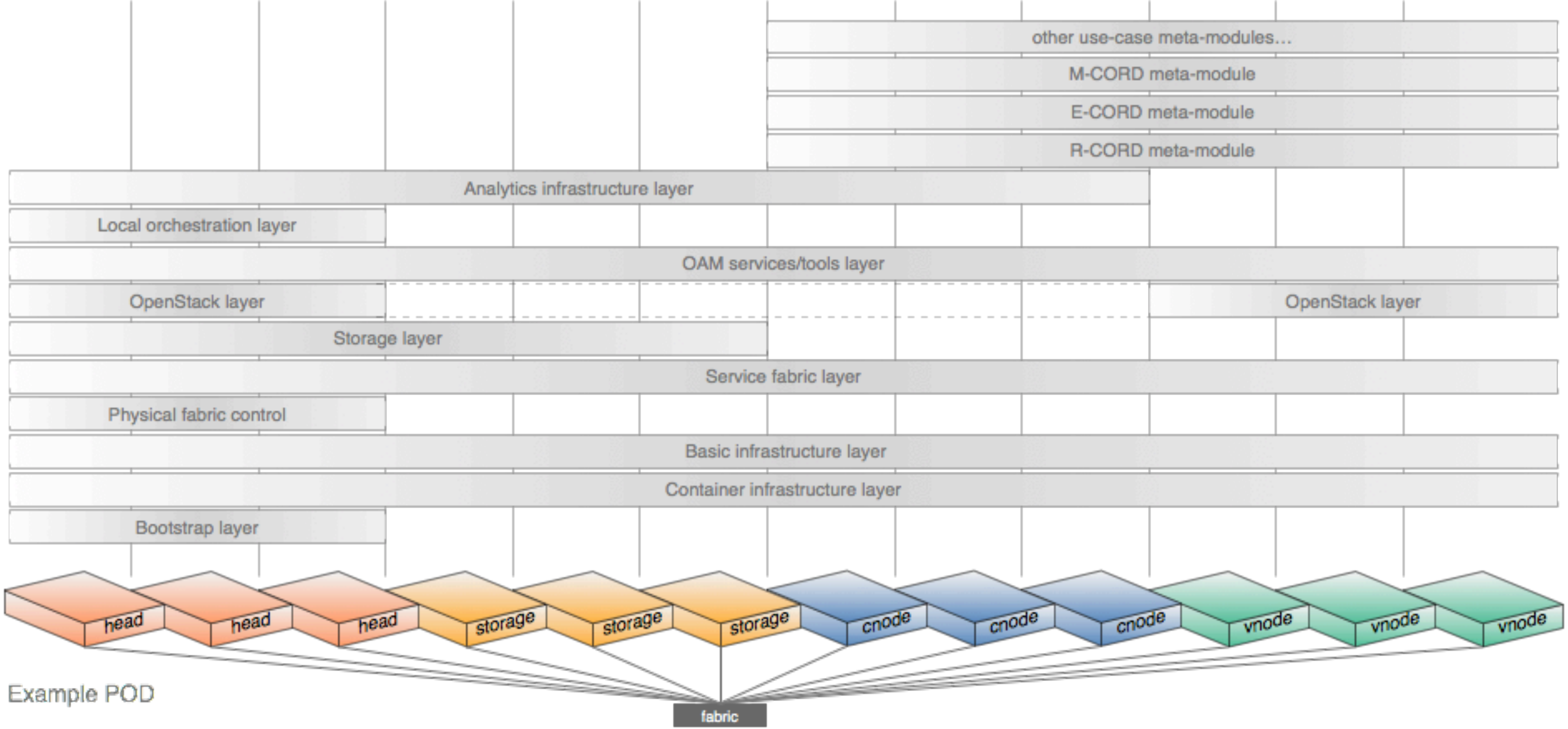
We will start with homogeneous server hardware and will specialize based on operational experience

Example:



#OpenCORD

Software Layers

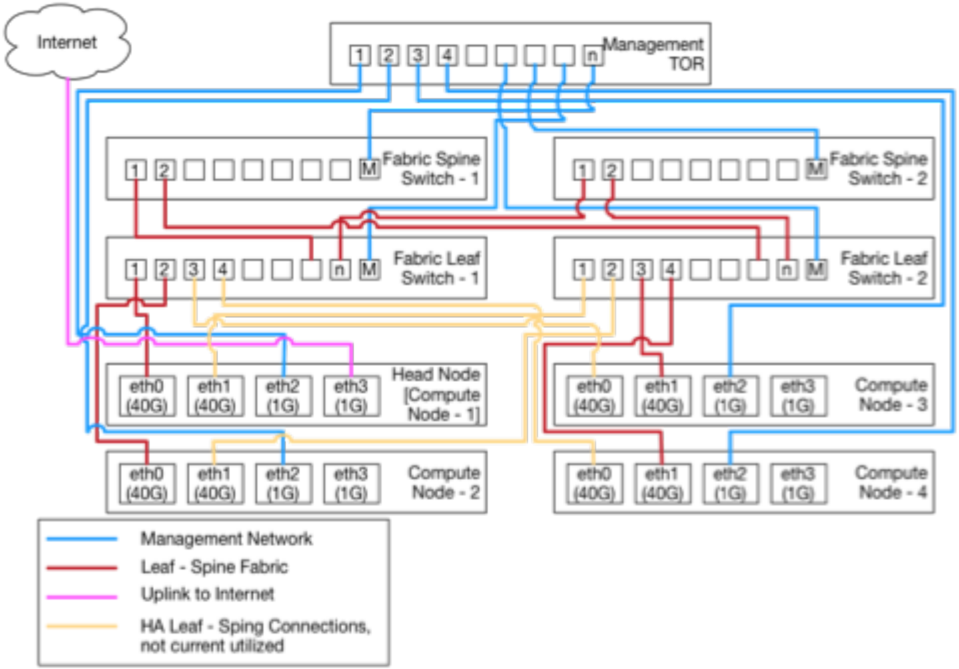


#OpenCORD

Deployment Model

1. All artifacts are buildable in a Vagrant box
   a. Ensures repeatability and consistency of experience across different environments
2. Artifacts are then published to the head-node
   a. Docker artifacts are pushed to a docker registry
   b. Maven artifacts are push to a maven repo
   c. Software packages are pushed to a apt repo
3. The remainder of the POD’s software and configuration is primed from the head-node
   a. Preserving consistent operation and deployment as well as a controlled versioning of software elements
   b. Also makes POD self contained without the need of an internet connection

9 Steps to POD*


Internet
1 2 3 4 n Management TOR
1 2 M Fabric Spine Switch - 1
1 2 M Fabric Spine Switch - 2
1 2 3 4 n M Fabric Leaf Switch - 1
1 2 3 4 n M Fabric Leaf Switch - 2
eth0 (40G) eth1 (40G) eth2 (1G) eth3 (1G) Head Node [Compute Node - 1]
eth0 (40G) eth1 (40G) eth2 (1G) eth3 (1G) Compute Node - 3
eth0 (40G) eth1 (40G) eth2 (1G) eth3 (1G) Compute Node - 2
eth0 (40G) eth1 (40G) eth2 (1G) eth3 (1G) Compute Node - 4
Management Network
Leaf - Spine Fabric
Uplink to Internet
HA Leaf - Sping Connections, not current utilized

1. Clone CORD Repository
   `git clone`†
1. Fetch Standard Images
   `./gradlew fetch`‡
1. Build Images‡
   `./gradlew buildImages`
1. Prime Seed Server*
   `./gradlew prime`
1. Publish Artifact to Seed Server
   `./gradlew publish`‡
1. Deploy PXE/DHCP/DNS
   `./gradlew deployBase`*
1. Deploy XOS
   `./gradlew deployPlatform`*
1. Turn up other resources
   `power on`
1. Verification(not yet implemented/ integrated)
   `./gradlew verify`

† - git, ‡ - docker, * - ansible

And the * is because…..

Configuration of Virtual Tenant Networking is manual

Configuration of Fabric is manual

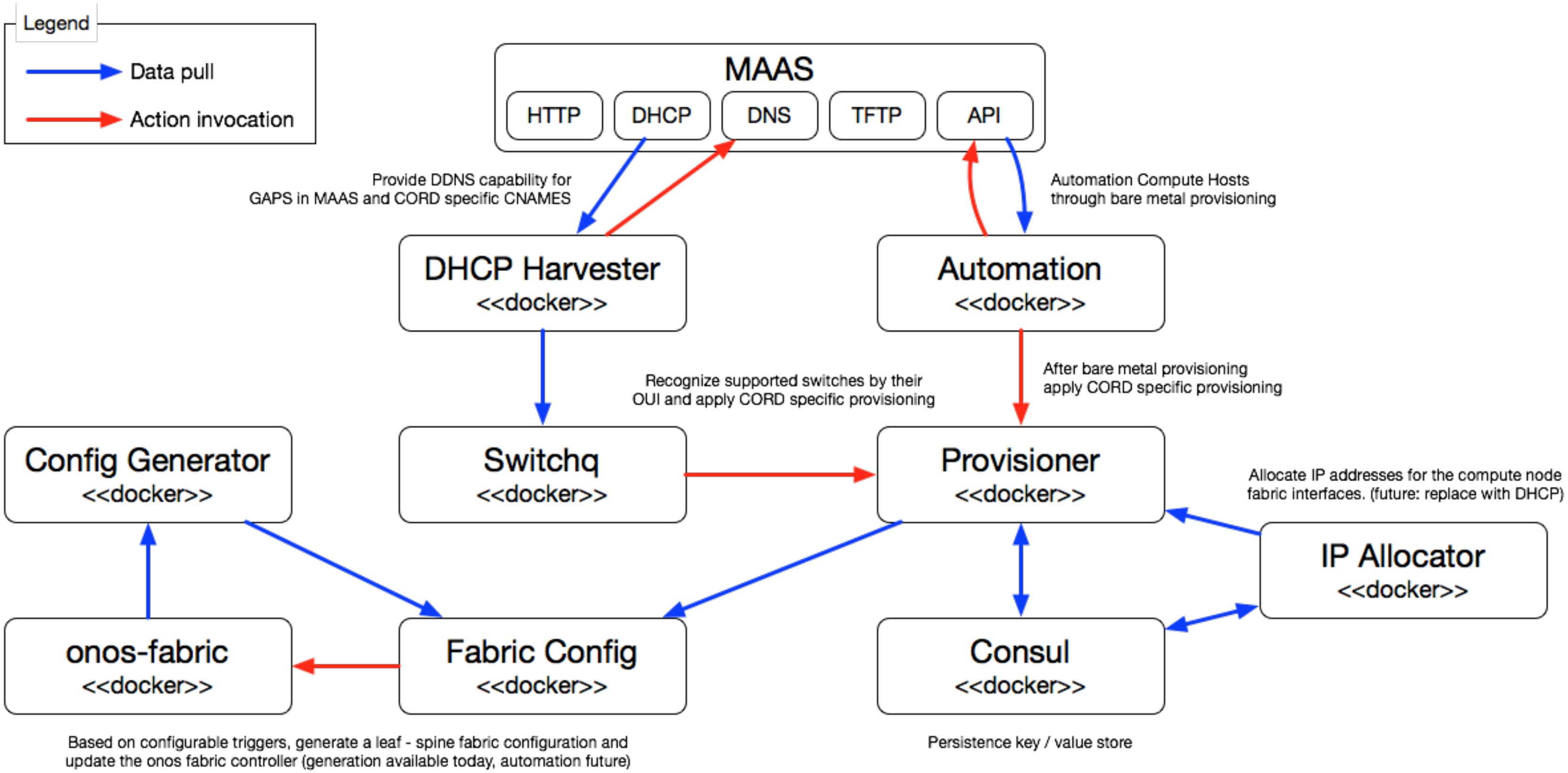
This one has a helper but it not yet automated

Configuration of Access Devices is manual

Configuration of vRouter is also a manual step.

Hopefully all this will be automated in future release

Automation μServices



Provide DDNS capability for GAPS in MAAS and CORD specific CNAMES

Automation Compute Hosts through bare metal provisioning

Recognize supported switches by their OUI and apply CORD specific provisioning

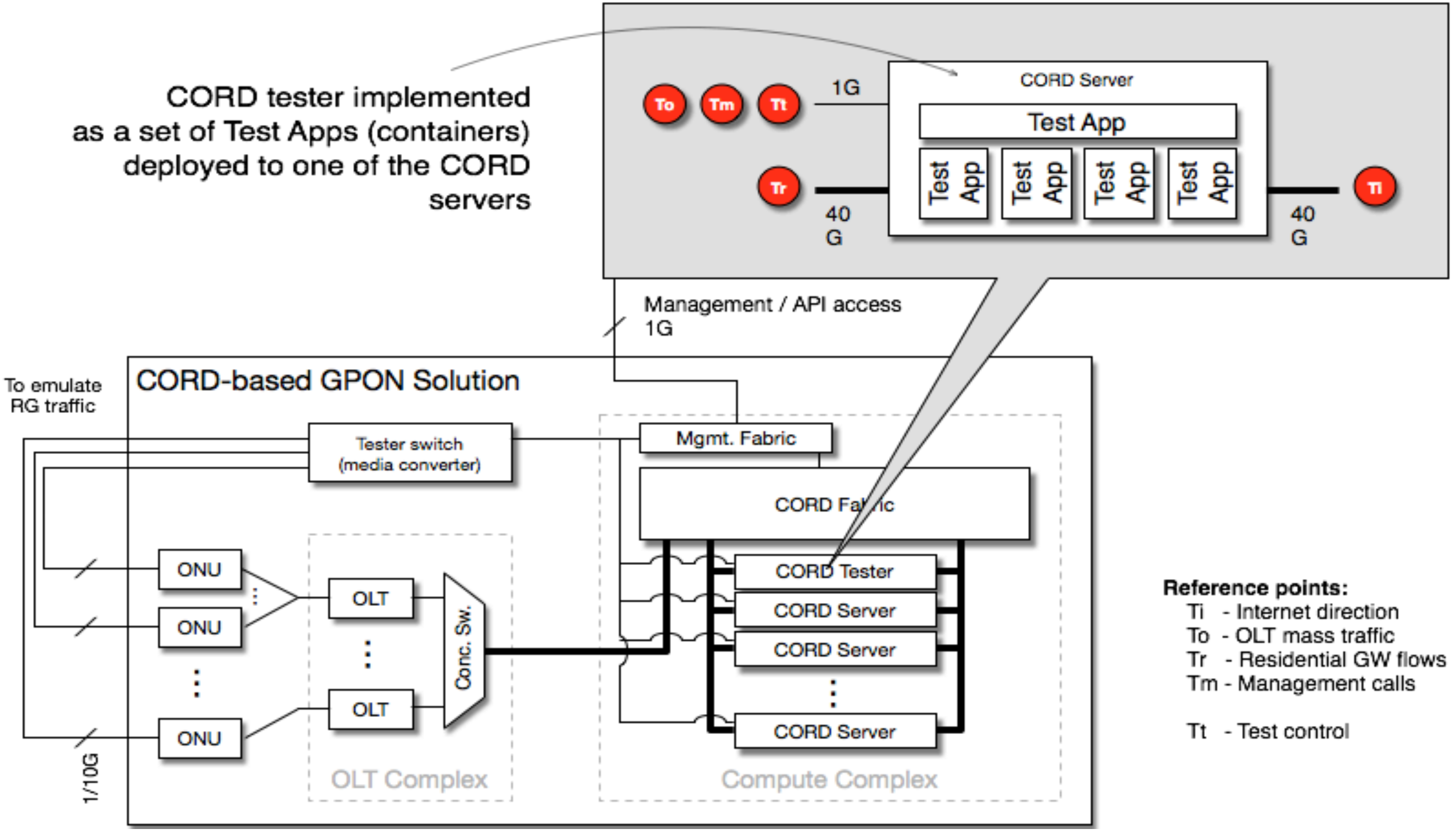
After bare metal provisioning apply CORD specific provisioning

Allocate IP addresses for the compute node fabric interfaces. (future: replace with DHCP)

Based on configurable triggers, generate a leaf - spine fabric configuration and update the onos fabric controller (generation available today, automation future)

Persistence key / value store

Automated Testing



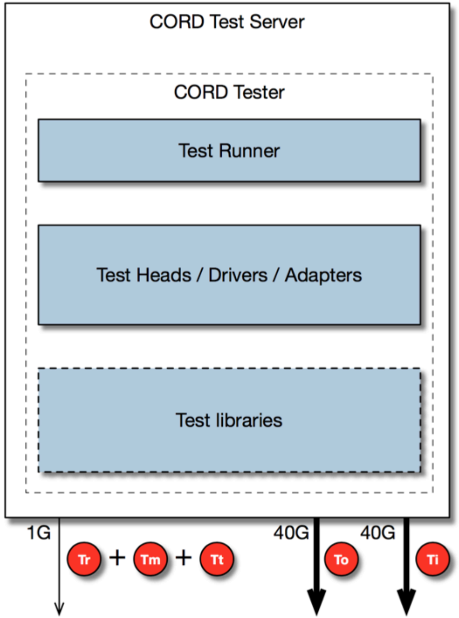
Reference points:

Ti - Internet direction
To - OLT mass traffic
Tr - Residential GW flows
Tm - Management calls

Tt - Test control

#OpenCORD

Automated Testing



Test roles per reference point:

Tr **Emulate a few residential gateways. Includes:**
- Authentication, DHCP/DNS
- Multicast participation (channel surfing + receiving streams)
- Internet access

Tm **Emulate service management flows. Represents the centralized Orchestrator/Controller/OSS. Includes:**
- Subscriber management
- Service instantiation
- Provide Radius test server
- Read service status
- Receive async events/callbacks (KPI, Alarms, etc.)

Tt **Test-specific control of the infrastructure. Includes:**
- Media converter management (VLANs)
- Any gray/white box probing (e.g., validation via flow entries and flow counters, direct API access for low level functions, etc.)

To **Emulate mass OLT traffic to mimic 100s or 1000s of subscribers. Includes:**
- Mass Internet access
- Receive large number of multicast streams

[- Lots of concurrent multicast join requests]

… all encapsulated as if coming from RGs via OLTs

Ti **Emulate the Metro network and the Internet. Includes:**
- Fine-grained flow emulation for few select subscribers
- Coarse-grained flow emulation for 100s/1000s of subscribers
- Provide multicast feeds

Comprehensively Testing CORD

Subscriber : Tests channel zap time , channel surfing experience, join/leave functions and latency , join/jump channel functions and latency, join/next channel functions and latency, duplicate joins , duplicate leaves, emulates 100s/1000s of Channels and subscribers for TLS Authentication, DHCP address assignment and subscriber traffic validation .

vRouter : Tests integration of Quagga container and ONOS vrouter app with multiple hosts. Tests different scales of route entries getting synched from Quagga to onos and getting applied as flows. Validating flows passing traffic.

TLS : It supports full fledged TLS client. Tests user traffic with AAA app using RADIUS server, TLS certificate based authentication.

IGMP : Tests protocol level join, leave , query messages with different options.

DHCP Server/Relay : Tests expected functionality and stresses by assigning large number of ip addresses to subscriber . It tests DHCP relay app with ISC dhcp server running outside in a container. It tests dhcp server onos app also as per standards.

Flows : Tests all kinds of flows applied by ONOS to switch. It validates flow implementation of ONOS by passing traffic to switch.

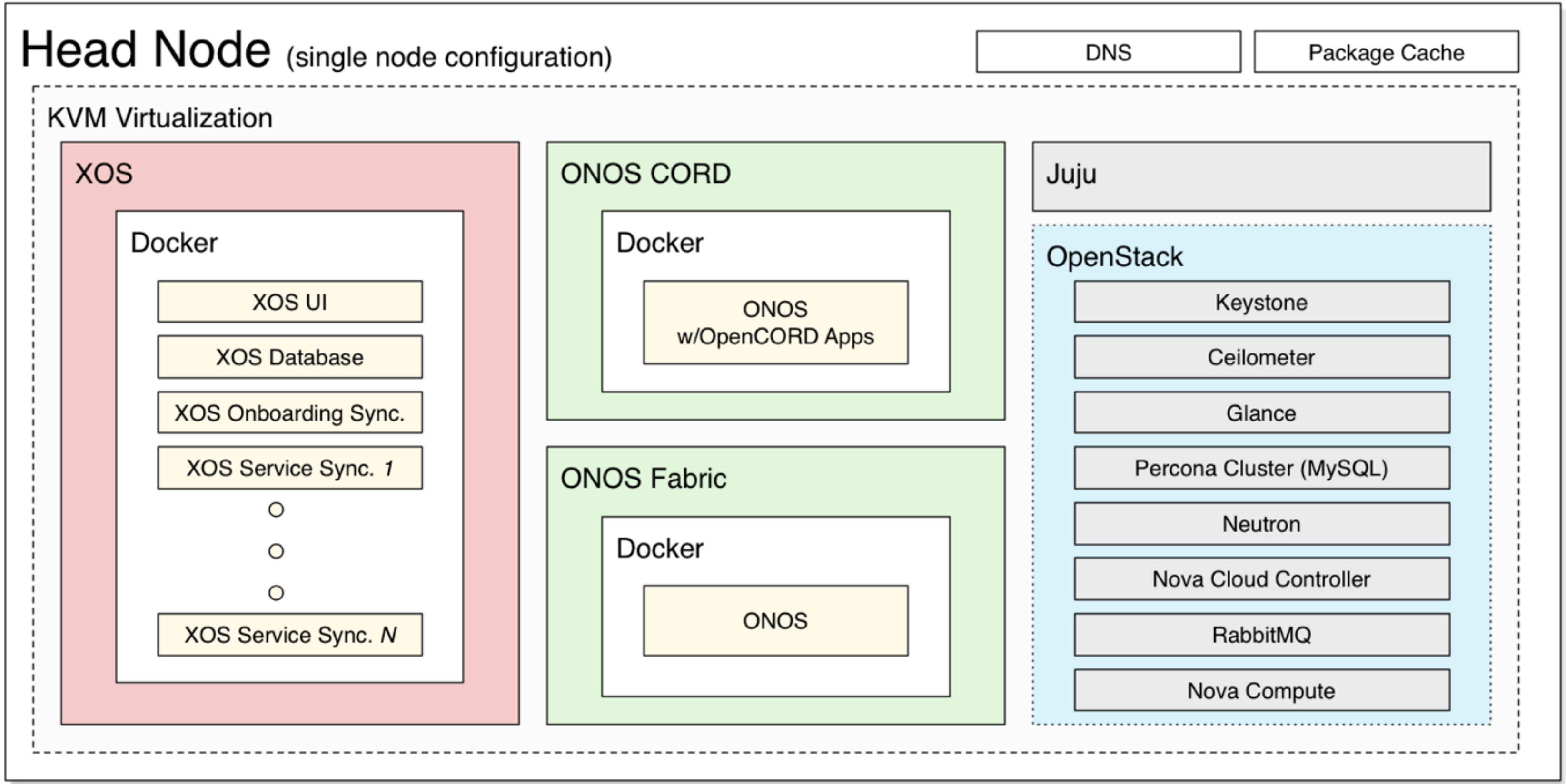
Zack Williams

Onboarding Services into CORD

CORD Installation Process

1. Nodes prepared (MAAS, or manual install)
2. Prereqs installed with platform-install
3. XOS services onboarded with service-profile

Single Node



Head Node (single node configuration)

DNS

Package Cache

KVM Virtualization

XOS

Docker

XOS UI

XOS Database

XOS Onboarding Sync.

XOS Service Sync. *1*

○

○

○

XOS Service Sync. *N*

ONOS CORD

Docker

ONOS w/OpenCORD Apps

ONOS Fabric

Docker

ONOS

Juju

OpenStack

Keystone

Ceilometer

Glance

Percona Cluster (MySQL)

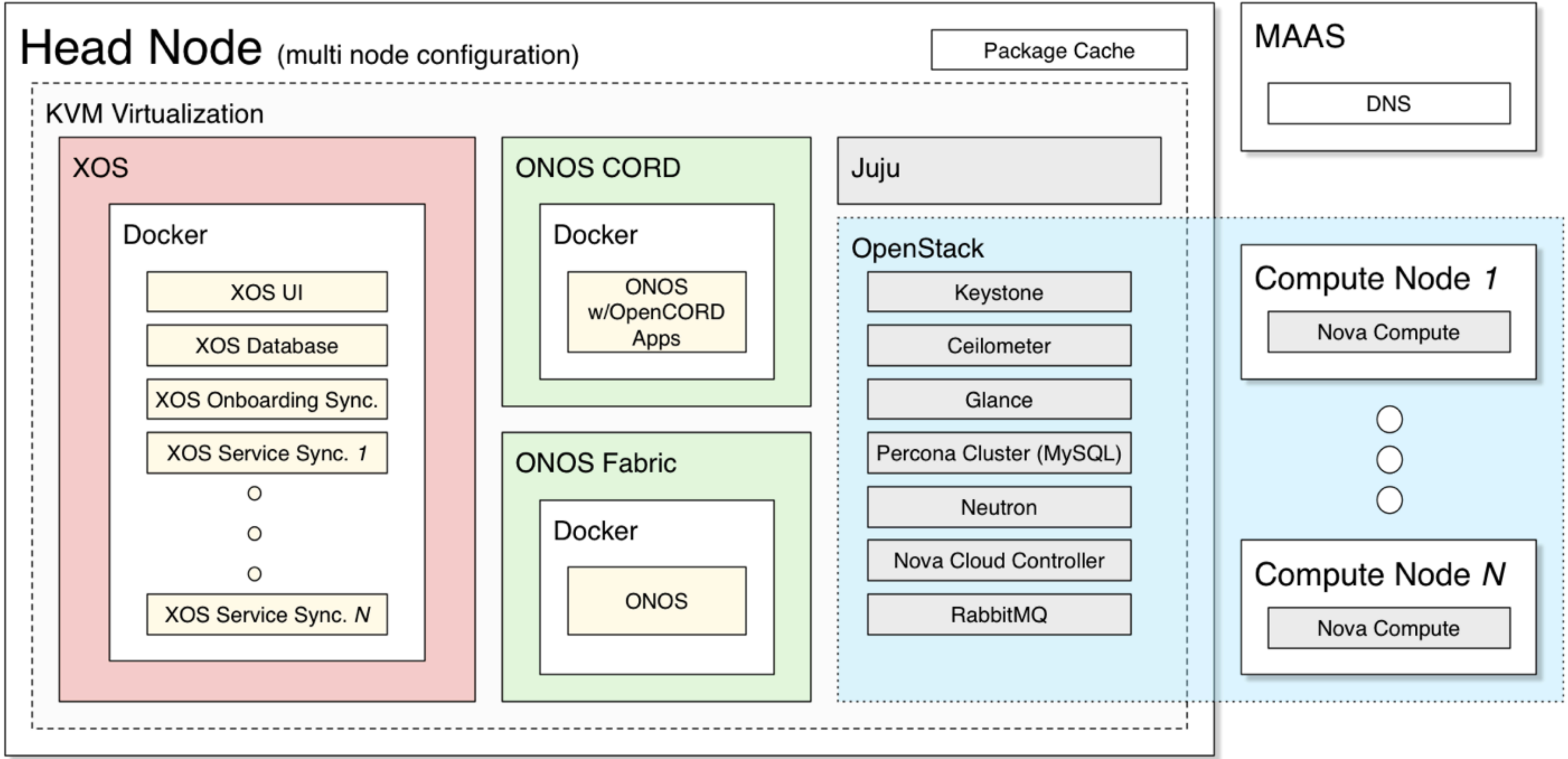
Neutron

Nova Cloud Controller

RabbitMQ

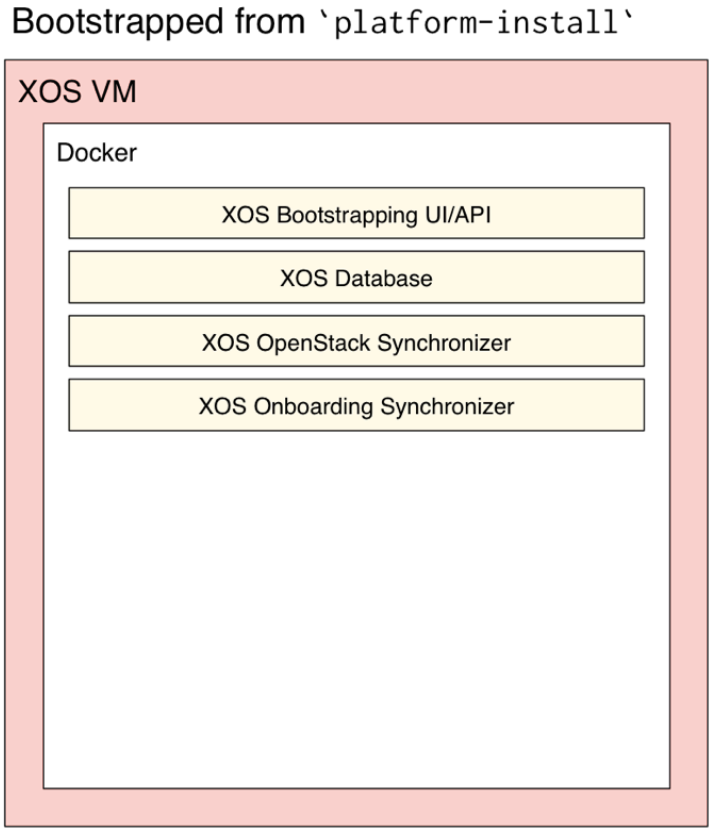
Nova Compute

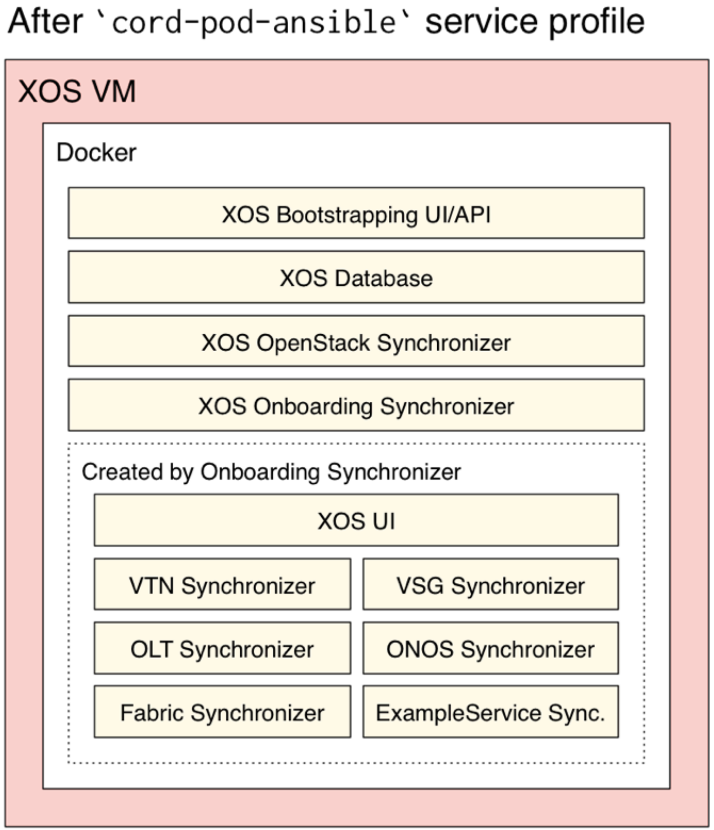
Multi Node



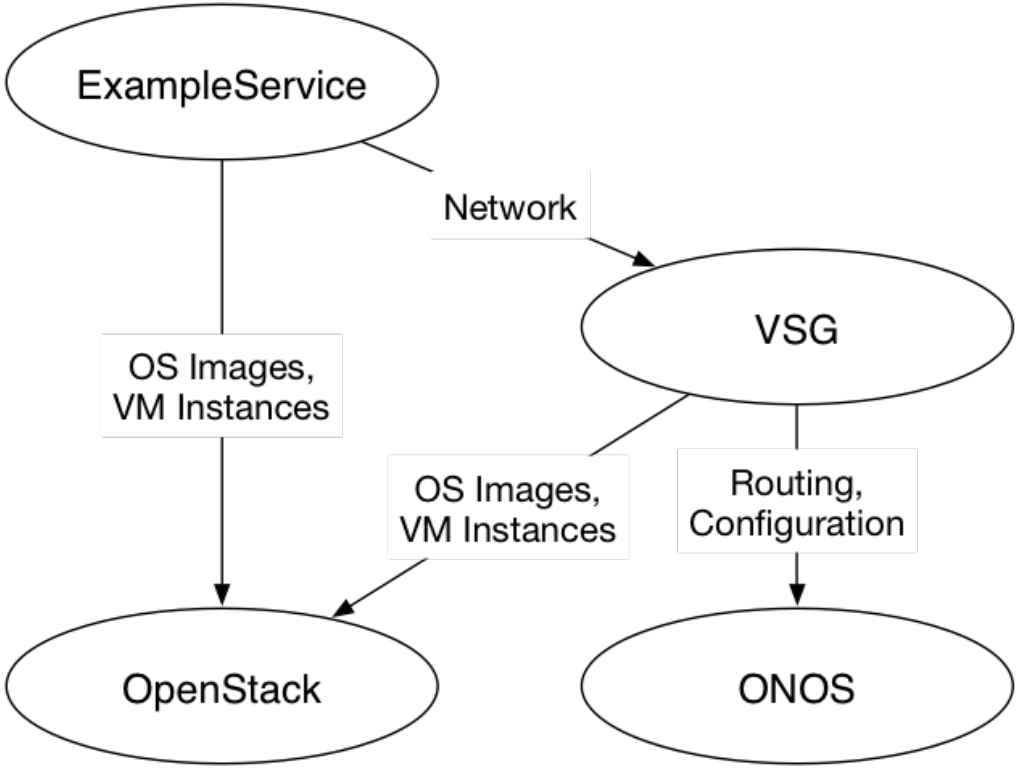
#OpenCORD

Service Profiles

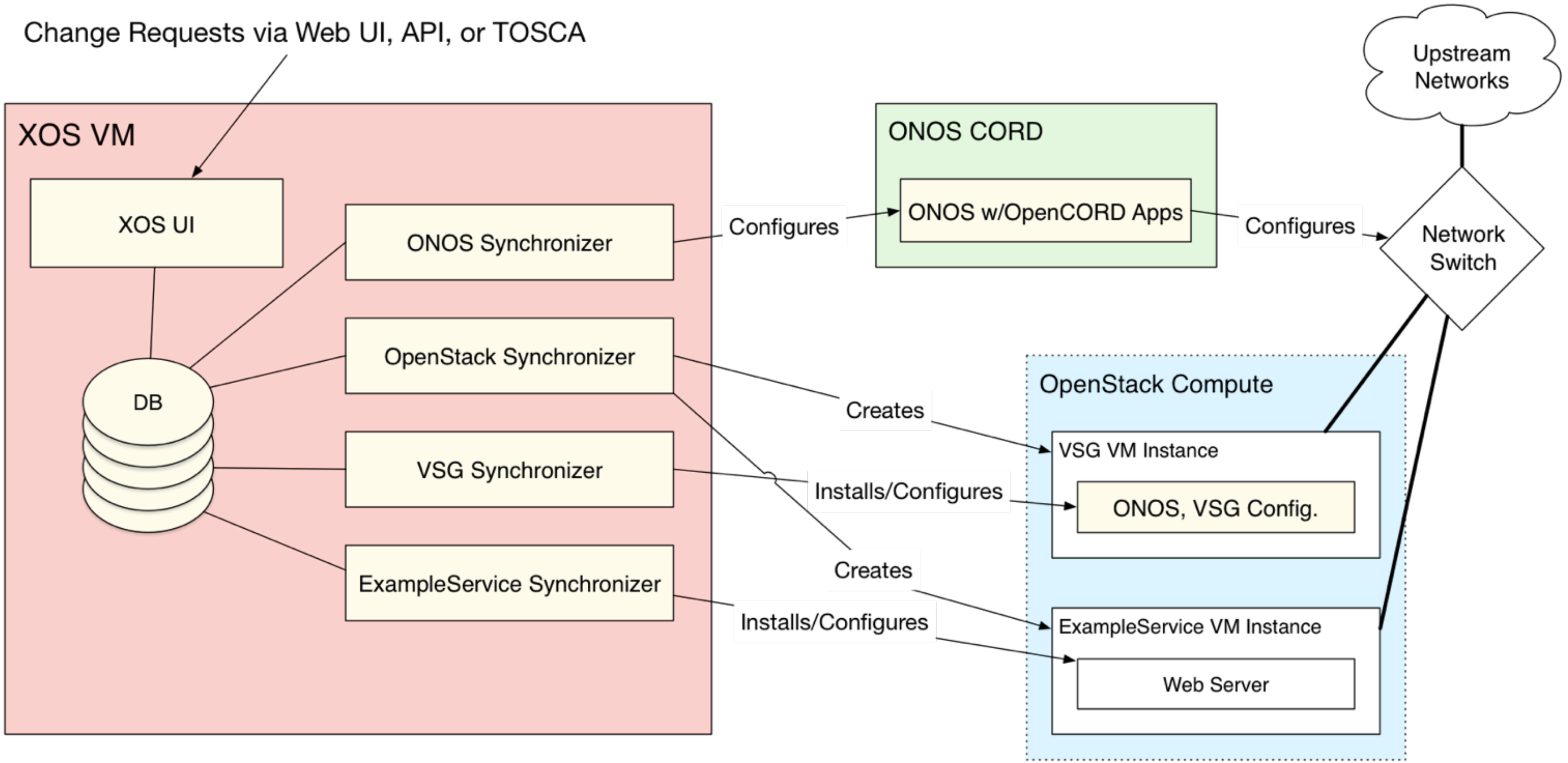

Bootstrapped from `platform-install`
XOS VM
Docker
XOS Bootstrapping UI/API
XOS Database
XOS OpenStack Synchronizer
XOS Onboarding Synchronizer



Service Dependency Graph



Synchronization Process



Anatomy of a Service Repository

Component	Code	Documentation
Onboarding Spec	xos/*-onboard.yaml	XOS
Data Model	xos/models.py	Django, XOS
Admin GUI	xos/admin.py	
REST API	xos/api/*	Django REST framework, XOS
TOSCA API	xos/tosca/*	XOS
Synchronizer	xos/synchronizer/*	Ansible, XOS
ONOS App	app/*, api/*	ONOS

Matteo Scandolo

Controlling CORD

About GUI, REST APIs and TOSCA

#OpenCORD

Controlling CORD

GUI

Visual Feedback

Diagnostic

REST APIs

Integration with other systems

External applications

On the flight configuration

TOSCA

System bootstrapping

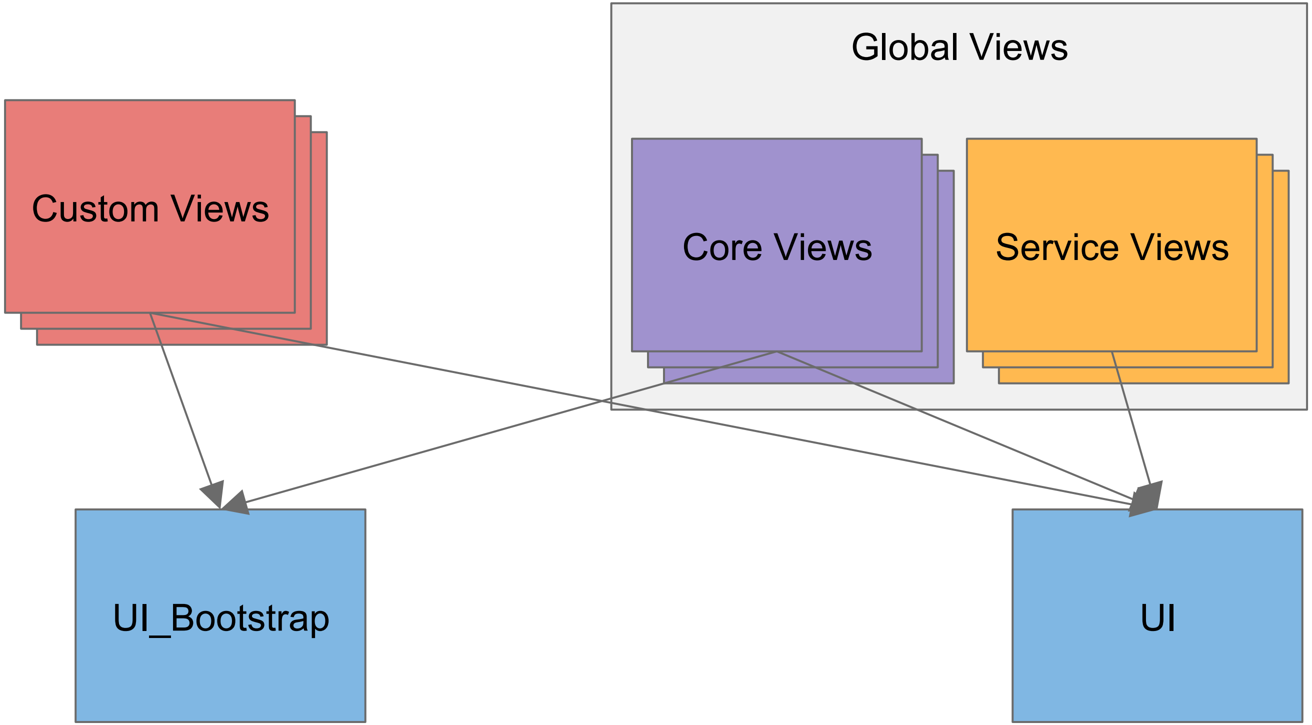
Service graph definition

Graphical User Interface

Composed by two pieces:

- Global Views
  - Powered by Django Admin
  - Core
    - Perform CRUD operations on CORE Models
    - Eg: Slices, Nodes, Networks, …
  - Services
    - Manage Services properties and related Tenants
    - Eg: vRouter, vSG, ...

GUI Structure



Note that this container is rebuilt any time a Service is onboarded

#OpenCORD

Define a Service View

Service View lives in the service repository (xos/admin.py)

Define all the views that are needed by your service

```
class VOLTServiceAdmin(ReadOnlyAwareAdmin):
```

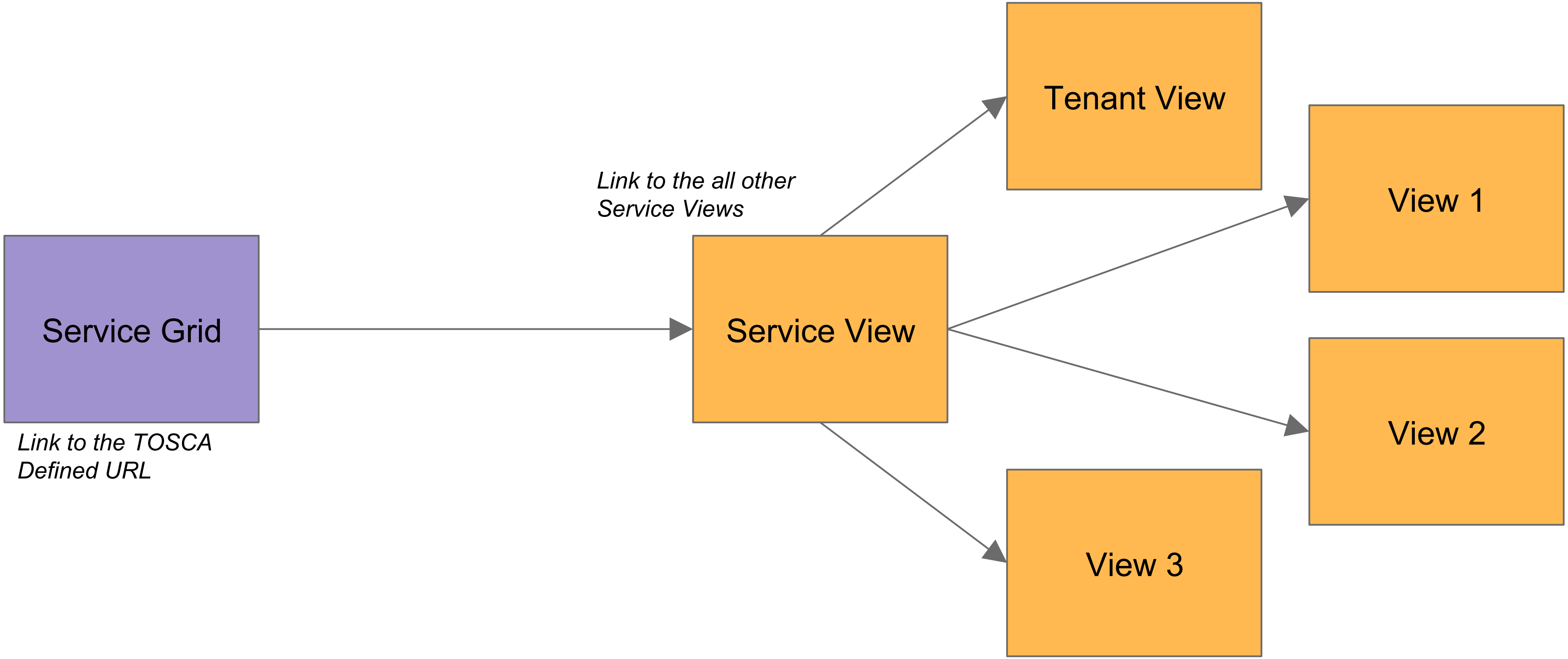
Register the view

```
admin.site.register(VOLTService, VOLTServiceAdmin)
```

Customize the service entry point with TOSCA

```
service#vtr:
     type: tosca.nodes.Service
     properties:
         view_url: /admin/vtr/vtrservice/$id$/
```

Service View Structure



Link to the all other Service Views

Link to the TOSCA Defined URL

Define a Custom View

Lives in the XOS Core (/views/ngXosViews/)

Loaded on demand

Take advantages of a UI Component Library (ngXosLib)

To create a new Custom View:

Create the application template

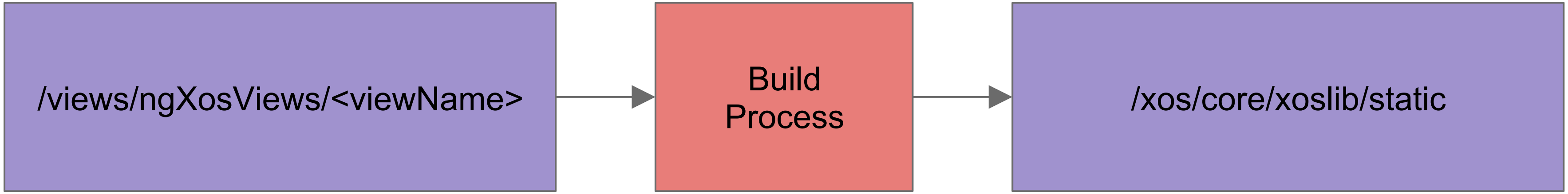
```
cd /views/ngXosViews/ && yo xos
cd <viewName> && npm start
```

Develop your features

Build the application

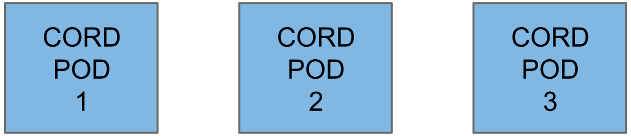
```
cd /views/ngXosViews/<viewName>/
npm run build
```

Custom Views Structure



Development Environment

- Source Files
- Dev Server
- Test Runner
- Handle different environment



- Minified JS
- Template Chaching
- Optimized CSS

What is ngXosLib?

A collection of UI Components:

Tables

Forms

Charts

…

Advantages:

Code Sharing

Fully tested

http://ngxoslib.wiki.opencord.org

#OpenCORD

REST APIs

- Follows REST (Representational State Transfer) Architecture:
  - JSON
  - HTTP Statuses
- Organized in four categories (prefixed by /api)
  - /core - Core models related API
  - /utility - Not related to a model, useful to manage the system
  - /service
  - /tenant

API Documentation

Located in /xos/tests/api

Use a Blueprint descriptive format (Markdown with custom syntax)

Published on **docs.xos.apiary.io**

- Can be used as a mock backend for development

Used to generate API Tests (Dredd)

API Usage Examples

API are used for:

- Custom Views
- External Applications
- Communication with services (eg: ONOS-CORD, ONOS-Fabric)

Usage examples:

https://github.com/opencord/xos/tree/master/xos/api/examples

TOSCA

Topology and Orchestration Specification for Cloud Applications

Used to describe:

- Data Model
- Services
- Service Dependencies

Loaded in the system through a dedicated REST Endpoint (/api/utility/tosca/run/)

Can be exported from the GUI

Thank you!

Questions?