

A group of birds flying in a V-formation against a light blue sky. The birds are silhouetted and appear to be in flight, with their wings spread.

# YANG Service Modeling

Jan Lindblad, Tail-f Business Development Solutions Architect

NSO Developer Days, June 2017

(c) 2017 Cisco. Cisco Public.

# Table of Contents

- Modeling top down or bottom up, which is better?
- Levels of Abstraction
- Stacked Services, Layered Service Architecture (LSA) and Reuse
- Managing External Resources
- Common YANG Modeling Mistakes

# Modeling a Service? How do You Start?

## Top Down

- Best models from customer point of view
- High abstraction may raise the bar for implementation
- Good in multivendor use cases

Start with the service YANG, look at devices, templates and code later

## Bottom Up

- Easiest way to start for a seasoned network engineer
- Low abstraction keeps mapping small and simple

Start with the device templates, figure out what in there needs to go in the YANG, then add any code if needed

## Shuttle

- In between results
- Start modeling from the top, then frequently alternate your top/bottom perspective until you have a complete model and mapping

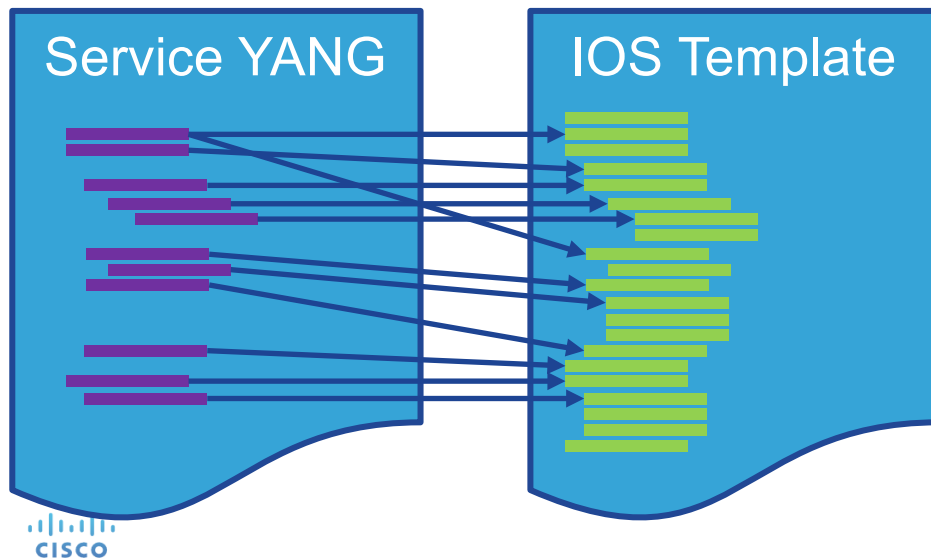
## Steal a Start

Stealing is both faster and easier. Quality?

# Abstraction

## Low abstraction

- Service model values copied to device by templates

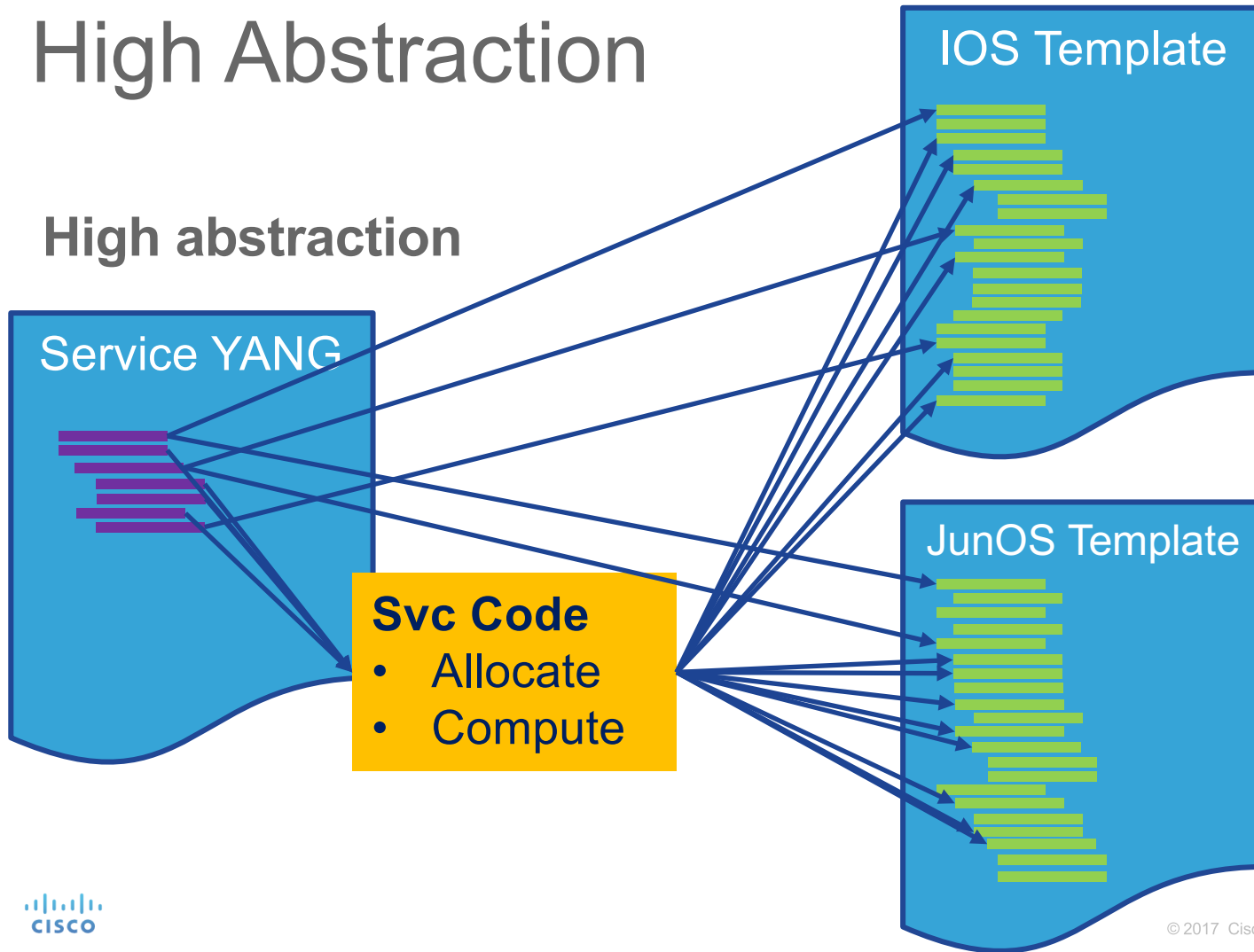


## High abstraction

- Service model is input data to code which computes values to device templates
- Many of the device model values are allocated/computed without human intervention

# High Abstraction

High abstraction



The service model contains high-level properties close to the customer's use case.

The customer's view here is quite abstract and needs to be realized by allocations and computations in code.

Since it is abstract, it works across many device types.

# Abstraction

## Low abstraction

```
leaf vlan-id {  
  type uint32 {  
    range "1..4094";  
  }  
}
```

```
<encapsulation>  
  <dot1Q>  
    <vlan-id>{/vlan-id}</vlan-id>  
  </dot1Q>
```

## High abstraction

```
vlan_id = alloc_vlan_id(zone)  
if(!vlan_id) return  
vars.add('VLAN_ID', vlan_id)  
template.apply(device_tmpl, vars)
```

```
<encapsulation>  
  <dot1Q>  
    <vlan-id>{$VLAN_ID}</vlan-id>  
  </dot1Q>
```

# Device Agnostic

Imagine a service supporting

CE: IOS

PE: IOS or IOS-XR

- Suddenly need to also support  
PE: JunOS

## Bottom Up Approach Service

- Service model YANG changes likely
- Easy to add a JunOS template

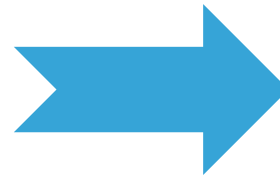
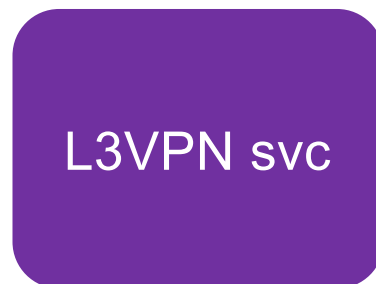
## Top Down Approach Service

- Service model YANG stable
- Need new code to handle device differences

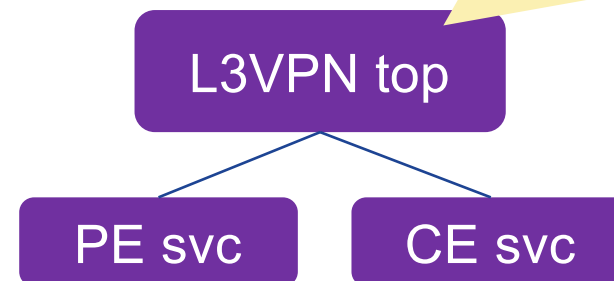
# Monolithic vs. Stacked Services

May feel natural to design an L3VPN service as a single monolithic service

Designing a service in several layers is known as stacked services. This might make the service easier to reuse and deploy



This is called a "stacked service"



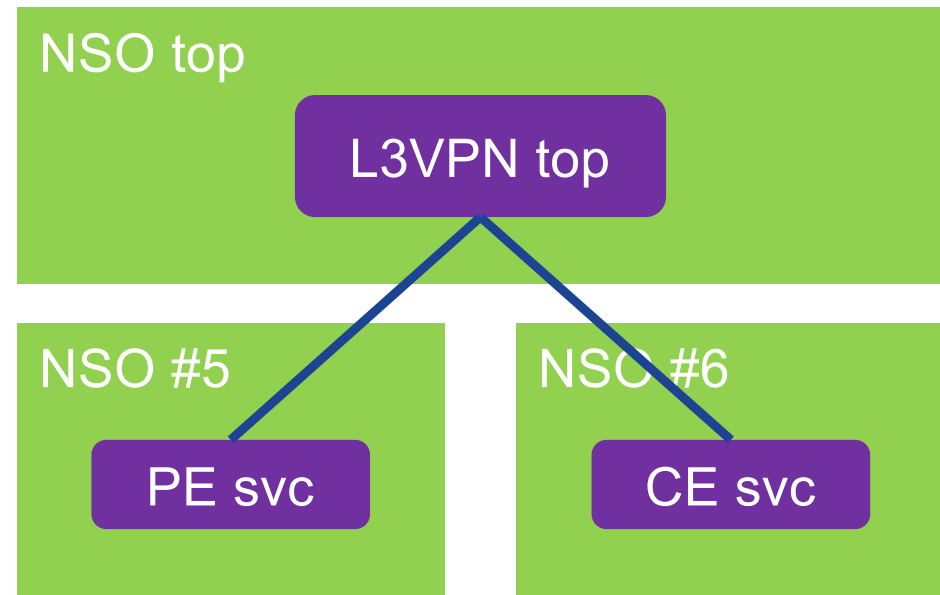


# Monolithic vs. Layered Service Architecture

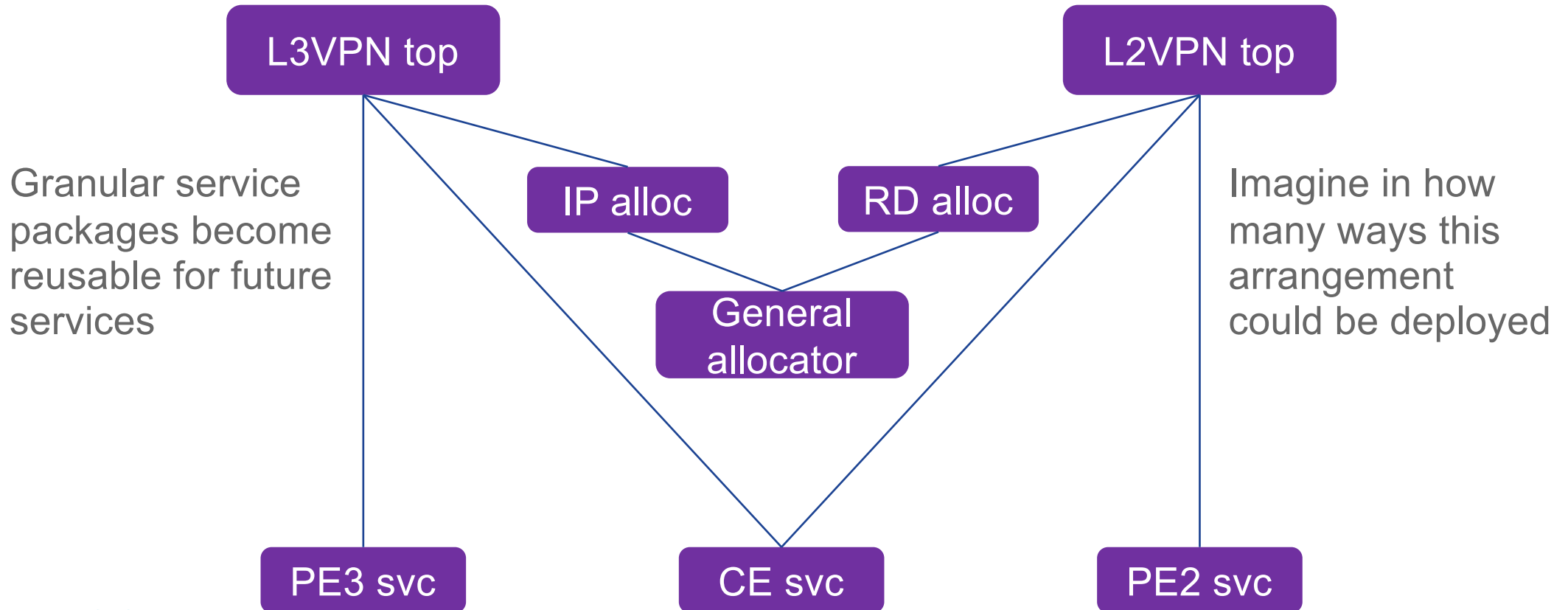


Because stacked services are more granular, they have more deployment options

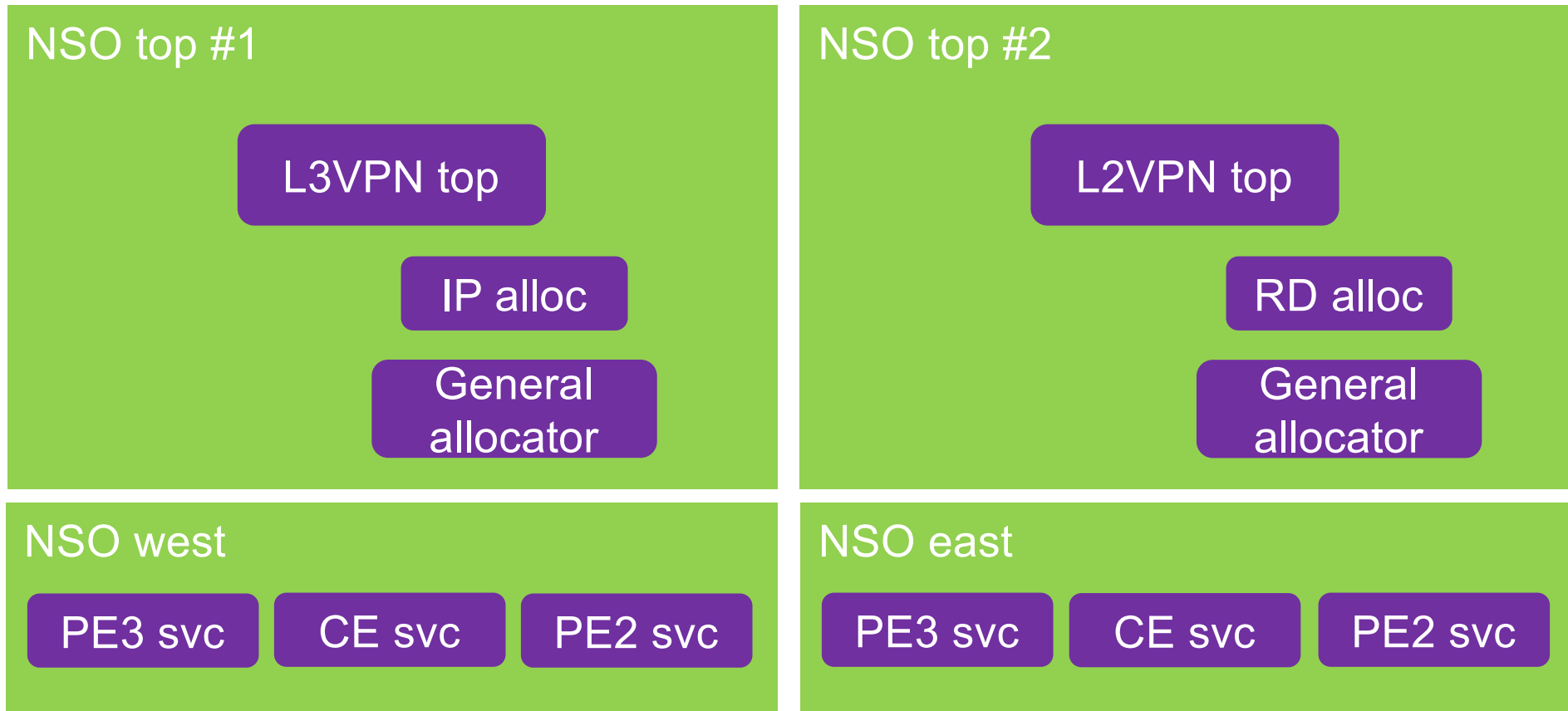
- A stacked service may be deployed in a more complex, parallel arrangement, which opens up for higher performance



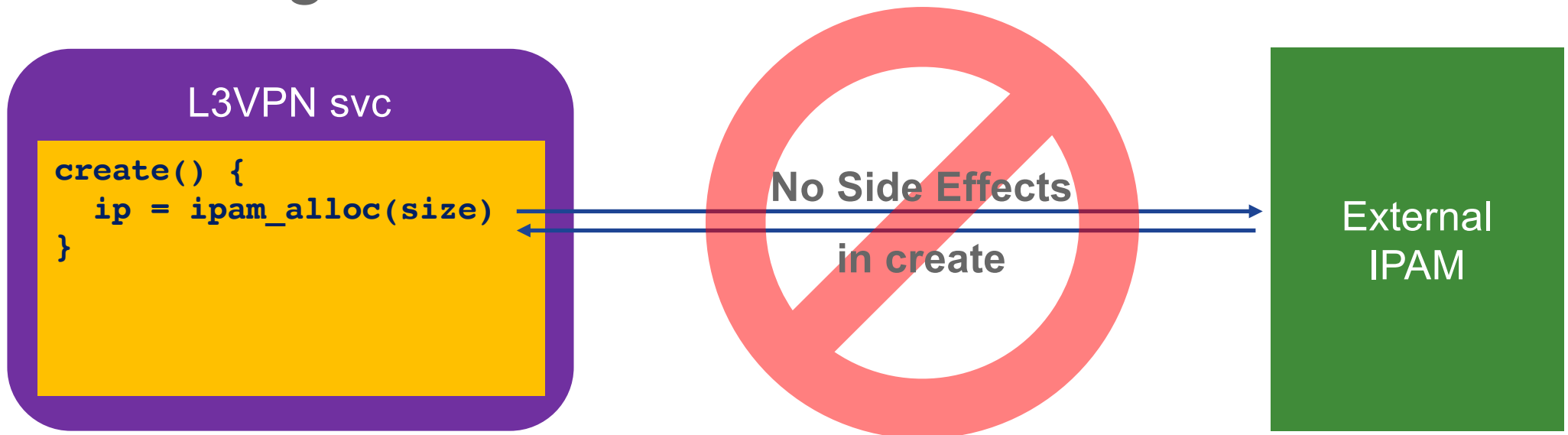
# Granular Services may Increase Reuse



# Granular Services Deployment Example



# Modeling External Resources



Directly calling an external entity in service create() will not work properly. The create() method needs to be "side effect free"

The create() method will be called during dry-run, the transaction commit might fail, and there is no delete() counterpart in a service

# Modeling External Resources

## L3VPN svc

```
create() {  
    ipam.req.size = 29  
    ipam.req.svc = me  
    if(!ipam.req.ip)  
        return  
}
```

Model resource ownership as configuration, and resource allocations as operational data

## IPAM sub

```
on_request(req) {  
    ip = ipam_alloc(size) →  
    req.ip = ip ←  
    req.svc.redeploy()  
}  
on_release(req) {  
    ipam_dealloc(req.ip) →  
}
```

Subscribers are ideal for handling external resources; they are notified at configuration create, modify and delete

External  
IPAM

# Modeling External Resources

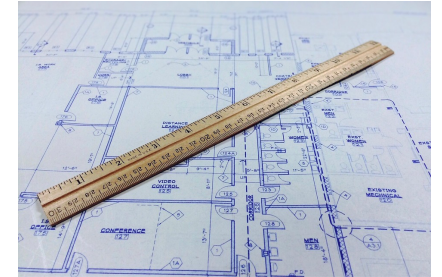
The resource-manager package is a plug-in framework for resource management

- Plug-ins may allocate different kind of resources
- Resources may be managed with/without external system
- Make your own plug-ins, or use resource-manager for inspiration



```
module ipaddress-allocator {
...
  augment "/ralloc:resource-pools" {
    list ip-address-pool {
      key name;
      uses ralloc:resource-pool-grp {
        augment "allocation/request" {
          leaf subnet-size {
            type uint8 {
              range "1..128";
            }
            mandatory true;
          }
        }
      }
    }
  }
...
}
```

# Tight YANG Models



YANG models are contracts  
between svc server/client

Client is building larger solution,  
not understanding the limits of  
the service will make the service  
hard to use

This is not the place to be fuzzy

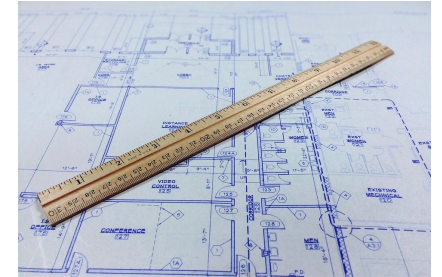
Imagine shopping at a website

You've evaluated and selected  
your products, passed through  
checkout and paid

Then you get a mail that the order  
is undeliverable?!

Will clients return to the shop?

# Tight YANG Models



## Clarity

- Service description, example(s)
- description vs. tailf:info
- mandatory true, default
- strings for arbitrary names, otherwise precision types

## Precision types

- range, length
- pattern
- leafref
- unique, max-elements, choice
- must, when

RFS pre-validation in NSO 4.4+



# Common YANG Mistakes: mandatory, default

Forgetting to add mandatory or default. Or at least a description

- leaf:s optional if nothing else specified

```
leaf enable {  
    type boolean;  
}
```

```
leaf mtu {  
    type uint32 {  
        range "68..65535";  
    }  
}
```

# Common YANG Mistakes: range, length

Forgetting to add simple ranges/  
lengths

- No-brainer to add
- Much value to client

```
leaf prefix-length {
    type uint8 {
        range "0..128";
    }
    description "IPv6 prefix length.";
}

typedef aaa_LdapFilter {
    type string {
        length "1..63";
    }
}
```

# Common YANG Mistakes: pattern

- Over-complicated patterns are unreadable and slow
- Not leveraging YANG 1.1 invert-match
- Using perl-regex, not W3C

```
pattern ""^((([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])\\.){3}([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])$|^((([a-zA-Z]|[a-zA-Z][a-zA-Z0-9\\-]*|[a-zA-Z0-9])\\.)*([A-Za-z]|([A-Za-z0-9\\-]*[A-Za-z0-9])$|^((?:[0-9a-fA-F]{1,4}){6}|(?:[0-9a-fA-F]{1,4}){4}|(?:[0-9a-fA-F]{1,4}){2}|[0-9a-fA-F]{1,4})))(?:[0-9a-fA-F]{1,4}){0,5})$";
```

```
pattern '*.example.com' {
  modifier invert-match;
}
```

```
typedef aaa_Email {
  type address_Email {
    pattern "^$|^((?!.{64,})[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+)$";
  }
}
```

# Common YANG Mistakes: leafref

```
leaf device {
  type leafref {
    path "/ncs:devices/ncs:device/ncs:name";
  }
  ...
choice interface-type {
  leaf ios-GigabitEthernet {
    when "deref(..../device)/../ncs:config/ios:interface/
          ios:GigabitEthernet/ios:name";
    type leafref {
      path "deref(..../device)/../ncs:config/ios:interface/
            ios:GigabitEthernet/ios:name";
    }
    ...
  }
  leaf iosxr-GigabitEthernet {
    when "deref(..../device)/../ncs:config/cisco-ios-xr:interface/
          cisco-ios-xr:GigabitEthernet/cisco-ios-xr:id" {
```

Use *leafref* to constrain input to usable values

Use *when* and *deref* to further constrain input to usable values

Good or bad to let device specifics shine through?

# Common YANG Mistakes: unique, max-elements, choice

Use unique, max-elements and choice whenever possible

- Easier to read+understand than must/when expressions
- Easier to write
- Faster validation

```
list tunnel {
  key "host port";
  unique local-port;
  // must "count(..../tunnel[local-port
  //           =current()]) = 1";

list name-server {
  min-elements 2;

...
}
// must "count(name-server) >= 2";

choice ipv4-or-ipv6 {
  leaf ipv4-address {...}
  leaf ipv6-address {...}
  // when "not(..../ipv4-address)";
```

# Common YANG Mistakes: must

- Not understanding when the expression applies
- Easy to make slow expressions
- Misunderstanding XPATH . (dot) node

```
container blocking {
  // presence "Enable blocked cmds";
  must "boolean(command)" {
    error-message "Command list must
      not be empty once initialized";
  }
  list command {

list tunnel { // 10, 1k or 1M entries?
  key "host port";
  must "count(../*tunnel[local-port
    =current()]) = 1";

leaf max {
  must "../*min < .";
  // must "../*min < current()";
```

