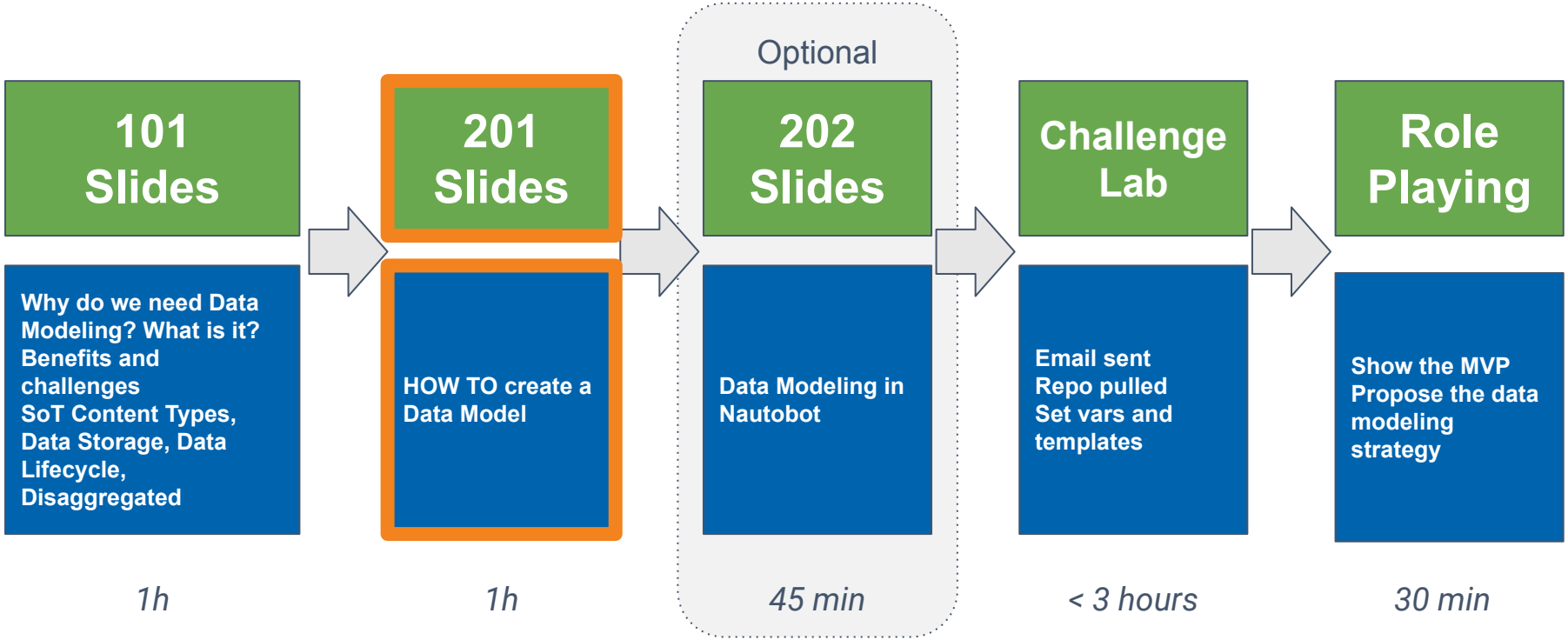
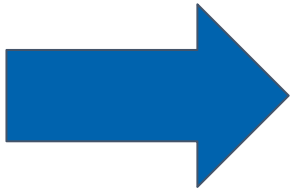

>>> network .toCode()
201 - Data Modeling
“HOW TO” Create a Data Model
Aug 16th, 2022

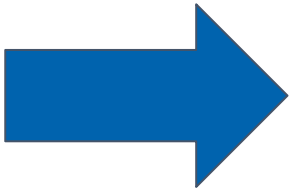
Data Modeling Training Plan



>>> Too many unknowns…



DATA MODEL



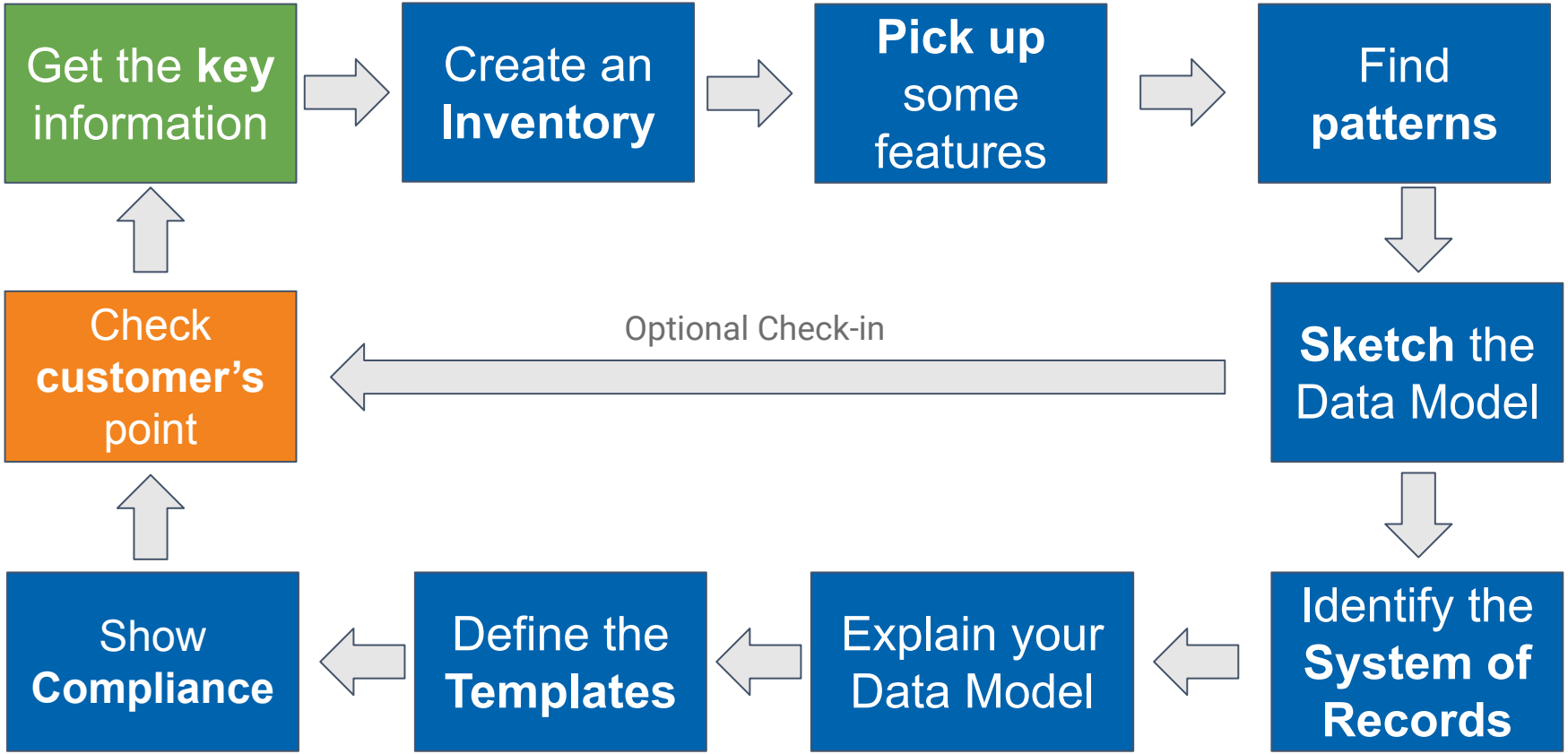
- **When is the data populated?** E.g. one time at the beginning, or for every new site
- **How is the data being populated?** E.g. UI, API, CSV?

- **Which features** these data represent?
- How many **different places** this data is stored in?

- **Who will consume** it? I.e., humans or automation
- **How** will it be **consumed**? What do the **templates** look like?

Data Modeling is a skill that comes with experience, do not get frustrated on the first try, soon you will start seeing the patterns, and your experience will help you to do more educated guesses

>>> The 10 steps to success *(hopefully)*



Get key information

Hint: Ask the right questions

>>> Get the key information

Maybe the most important step!

Describe all the steps to configure a network device

Obtain “golden configs” for each relevant grouping

Define the use-cases for the data model

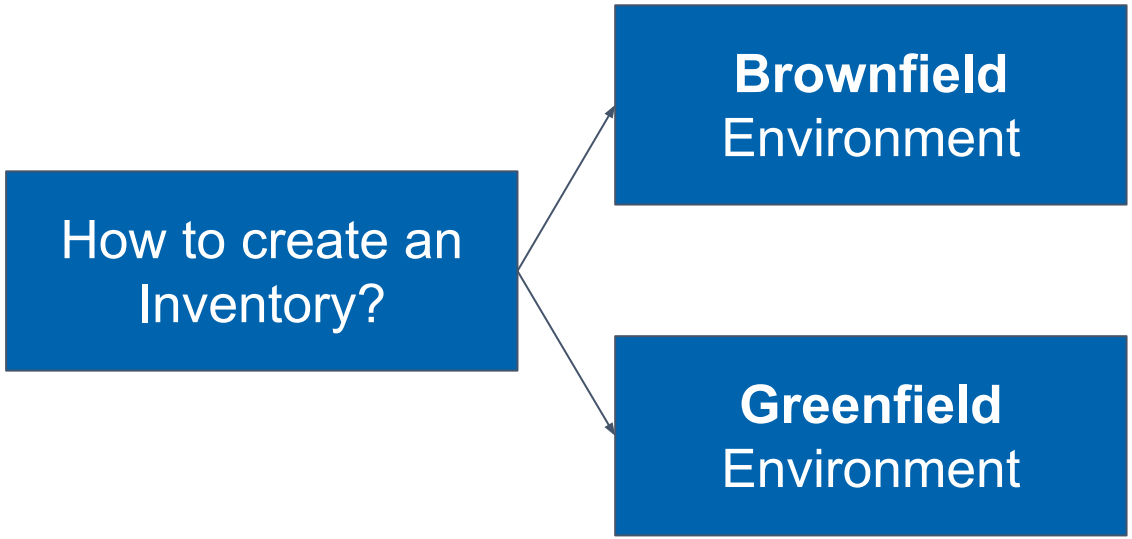
Check the **Workflow Discovery** training to learn about how to drive this conversation

- Is there any **documentation** (spreadsheets, diagrams, etc.) describing which data goes to each device?
- **Find any dependency** with existing automation, or external sources of information (e.g. an IPAM system)
- **Get real configurations from devices,** closer to the standard one, and for each type of platform, region, role, etc.
- Try to understand what is **per-design** vs what has been added as **snowflake**
- Understand if the **final goal** is to generate a full configuration for configuration provisioning, or a partial one for configuration compliance
- **Understand the scope** of the data model, which part of the network is targeted

Create an Inventory

Hint: The tallest buildings start with a single brick

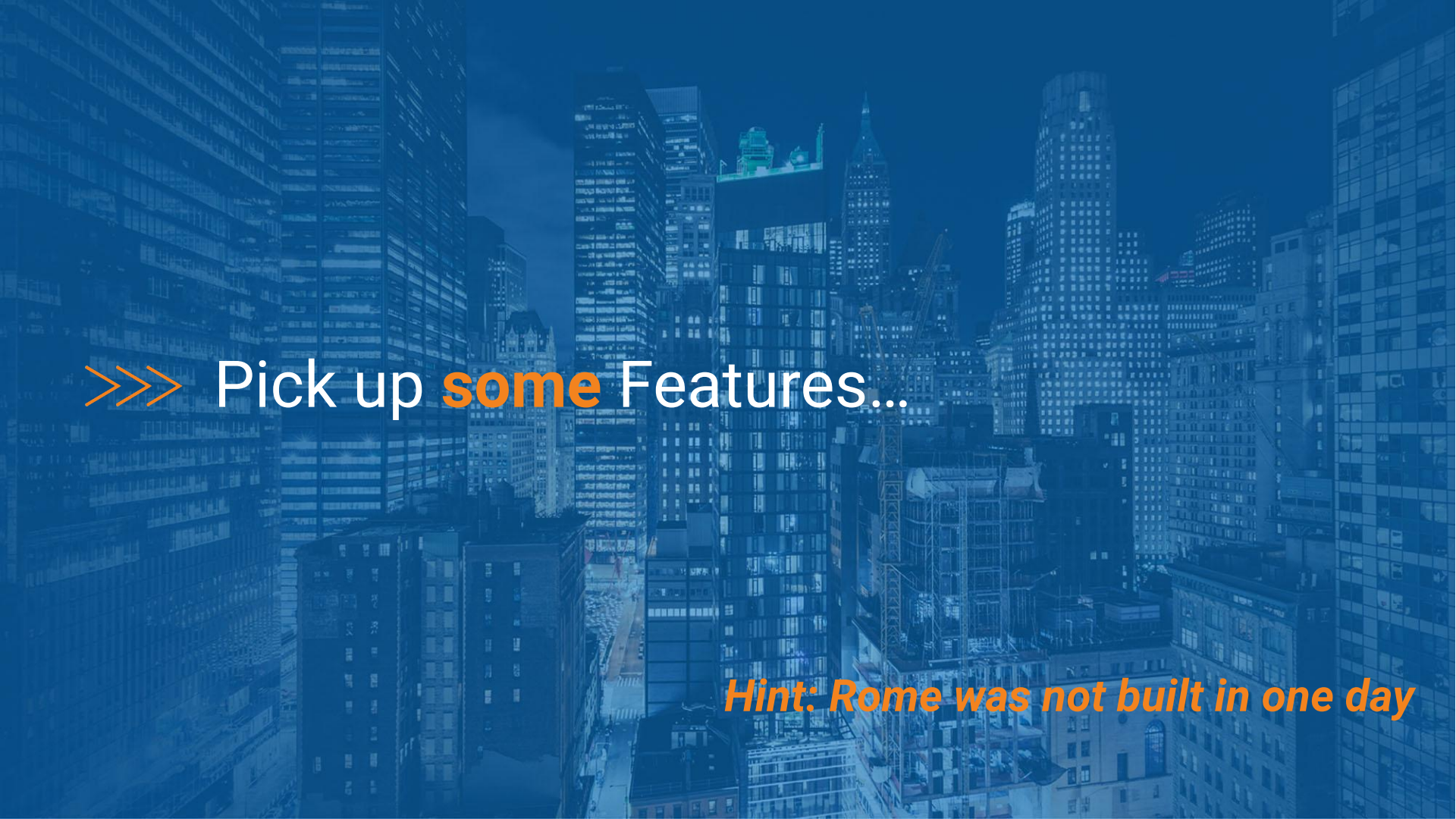
>>> Create an Inventory



Use network automation **discovery tools (e.g. Device Onboarding, Network Importer)** or **from data (CSV, excel, etc.)**

Use **design builder solutions** to translate from network design to inventory data

The Network Inventory is the foundation of the network automation strategy because it defines the **SCOPE** of the automation, and the **KEY** parameters to **classify the data**, and to access the network resources

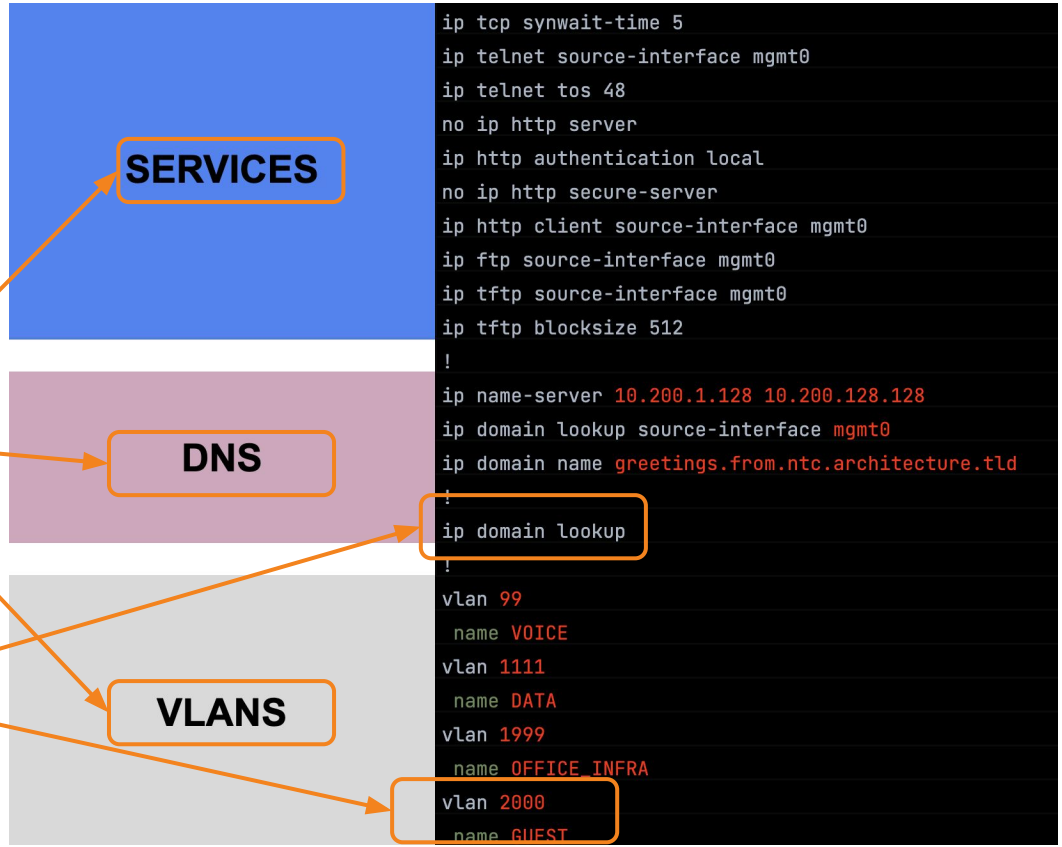

Pick up some Features...
Hint: Rome was not built in one day

>>> Start with Config Maps

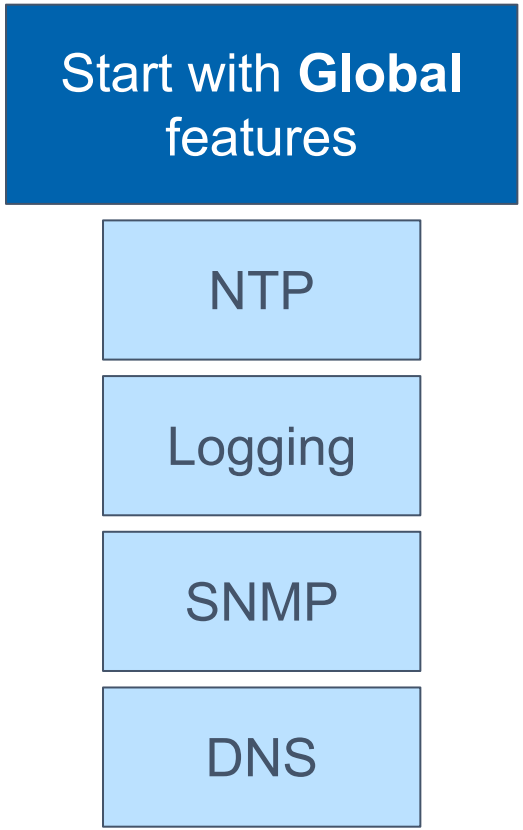
A **Config Map** helps identifying the relevant part of the config, and the key data

Classify the **config snippets** into **Feature Sections,** which define a configuration feature

Get a first guess of what looks like **static** vs **variable** data



>>> Which features are better to start with?



- Easier to understand
- Easier to model and implement
- Apply to most of the network devices
- **Start delivering value!**

As we target new features, it gets more complicated, and other than inventory data will be needed…


Find patterns -> Levels of Intent

>>> Disaggregating Configuration from Data Model

Complexities of Data Modeling

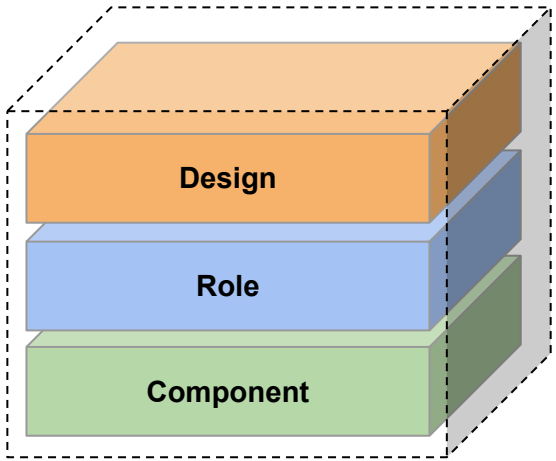
```
interface Ethernet1
  switchport access vlan 150
  switchport mode access
  authentication open
  authentication order dot1x mab
  authentication priority dot1x mab
  authentication port-control auto
  authentication periodic
  authentication timer reauthenticate server
  authentication violation replace
  mab
  dot1x pae authenticator
  dot1x timeout tx-period 8
  spanning-tree portfast
```

```
interface:
 - name: Ethernet1
   vlan: 150
   switchport_mode: access
   authentication: open
   authentication_order: [dot1x, mab]
   authentication_priority: [dot1x, mab]
   authentication_port-control: auto
   authentication_periodic: true
   authentication_timer_reauthenticate: server
   authentication_violation: replace
   mab: True
   dot1x_pae: authenticator
   dot1x_timeout: tx-period 8
   spanning-tree_portfast: True
```

```
interface:
 - name: Ethernet1
   vlan: 150
   dot1x: True
   spanning-tree_portfast: True
```

Don't map one-for-one configuration, provide the intention in the data of the configuration

Multiple Levels of Intent



Compression of data is key to any good data modeling strategy

Providing multiple levels of intent is one of those strategies.

>>> Multiple Levels of Intent - Example

Design

```
/leaf-spine_v1.yml
leaf:
  ports:
    uplinks: Ethernet1-4
    servers: Ethernet5-48
  device_vlans: [10, 15, 20]
```

Role

```
/roles.yml
interface_roles:
  uplink:
    mtu: 1500
    mode: trunk
    vlan: {{ device_vlans }}
```

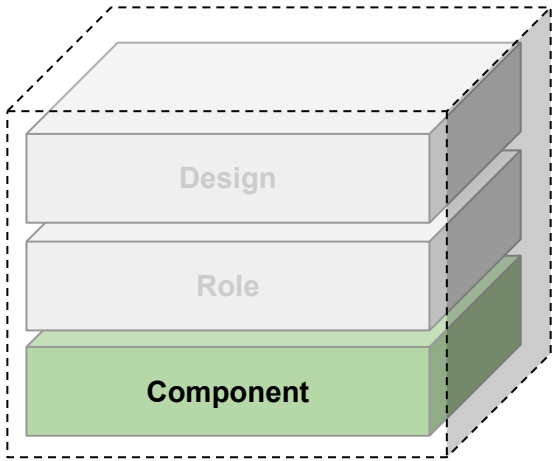
Component

```
/rtr-chi-01.yml
interfaces:
  Ethernet0:
    uplink_device: nyc-spine01
    uplink_interface: Tg0/0
  Ethernet5:
    server_name: nyc-apache01
    vlan: 10
```

```
Vlans: [10, 15, 20]
Interfaces:
  Ethernet1:
    uplink_device: nyc-spine01
    uplink_interface: Tg0/0
    mtu: 1500
    mode: trunk
    vlan: [10, 15, 20]
...
  Ethernet5:
    server_name: nyc-apache01
    mtu: 1500
    mode: access
    vlan: 10
...
```

Runtime Result

>>> Multiple Levels of Intent - Component

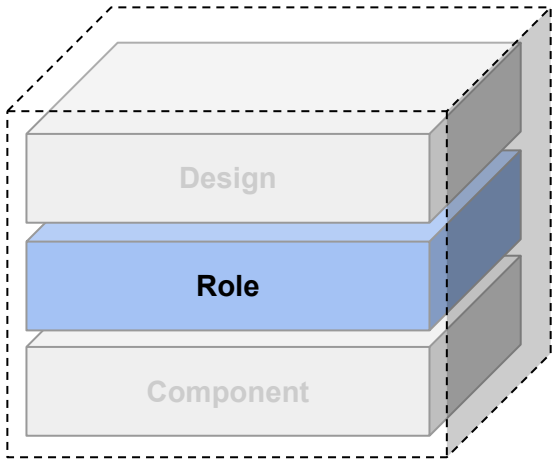


The component level is what is particular to a specific the device.

For instance, the name of the connected server

```
/rtr-chi-01.yml
interfaces:
  Ethernet0:
    uplink_device: nyc-spine01
    uplink_interface: Tg0/0
  Ethernet5:
    server_name: nyc-apache01
    vlan: 10
```

>>> Multiple Levels of Intent - Role



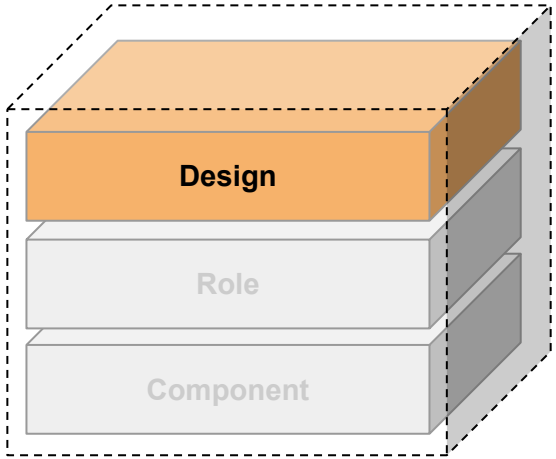
The role level captures what is identical across devices/interfaces of the same role.

All uplinks to aggregation must have a MTU of 1500

All uplink to aggregation must be a trunk mode, and use have all vlans

```
/roles.yml
interface_roles:
  uplink:
    mtu: 1500
    mode: trunk
    vlan: {{ device_vlans }}
```

>>> Multiple Levels of Intent - Design



The design level captures the intent of the network design.

The vlans of the trunk ports are deduced from a higher design, and the interface profile allocation of the "leaf" switches is defined by the design version.

```
/leaf-spine_v1.yml
leaf:
  ports:
    uplinks: Ethernet1-4
    servers: Ethernet5-48
  device_vlans: [10, 15, 20]
```

```
/leaf-spine_v2.yml
leaf:
  ports:
    uplinks: Ethernet1-8
    servers: Ethernet9-24
  device_vlans: [10-40]
```


Sketch the Data Model
Hint: Did you play to find Waldo?

>>> Infer the data model

```
vrf definition Blue
 rd 10.4.26.3:2047
 !
  address-family ipv4
  route-target export 64898:2047
  route-target import 64898:2047
 exit-address-family
!
!
logging count
logging userinfo
logging buffered 100000
no logging console
logging cns-events notifications
!
!
!
aaa new-model
!
aaa group server tacacs+ ACS
 server name ACS-1
 server name ACS-2
 ip vrf forwarding Blue
 ip tacacs source-interface Loopback2047
```

Configuration

Extract, per feature, the data that is **static (for template)** and **variable (for data model)**
Use your network engineer common sense!

From the variable data, later, we will understand which one is **per device**, or belongs to **some group membership**

Doesn't need to be 100% accurate, we will iterate on it as we discover more information

The Data Model **IS NOT** a full collection of all the configuration variables, only the **RELEVANT** ones in an **ABSTRACT** way
For instance, an OpenConfig model is not what we are looking for

>>> Infer the data model

```
vrf definition Blue
 rd 10.4.26.3:2047
 !
  address-family ipv4
  route-target export 64898:2047
  route-target import 64898:2047
 exit-address-family
!
!
logging count
logging userinfo
logging buffered 100000
no logging console
logging cns-events notifications
!
!
!
aaa new-model
!
aaa group server tacacs+ ACS
 server name ACS-1
 server name ACS-2
 ip vrf forwarding Blue
 ip tacacs source-interface Loopback2047
```

Feature: VRF

Feature: Logging

Feature: AAA

Configuration

>>> Infer the data model

```
vrf definition Blue
 rd 10.4.26.3:2047
 !
  address-family ipv4
  route-target export 64898:2047
  route-target import 64898:2047
 exit-address-family
!
!
logging count
logging userinfo
logging buffered 100000
no logging console
logging cns-events notifications
!
!
!
aaa new-model
!
aaa group server tacacs+ ACS
 server name ACS-1
 server name ACS-2
 ip vrf forwarding Blue
 ip tacacs source-interface Loopback2047
```

Feature: VRF

Feature: Logging

Feature: AAA

Configuration

>>> Infer the data model

```
vrf definition Blue
 rd 10.4.26.3:2047
 !
  address-family ipv4
   route-target export 64898:2047
   route-target import 64898:2047
 exit-address-family
!
!
logging count
logging userinfo
logging buffered 100000
no logging console
logging cns-events notifications
!
!
!
aaa new-model
!
aaa group server tacacs+ ACS
 server name ACS-1
 server name ACS-2
 ip vrf forwarding Blue
 ip tacacs source-interface Loopback2047
```

Feature: VRF

Feature: Logging

Feature: AAA

Configuration

Feature VRF:

```
vrf_name:        "Blue"
vrf_rd:          "10.4.26.3:1047"
vrf_export:      "64898:2047"
vrf_import:      "64898:2047"
```

Feature AAA:

```
aaa_servers:     ["ACS-1","ACS-2"]
aaa_source_int:  "Loopback2047"
aaa_vrf:         "Blue"
```

Data

How to represent a Data Model?

The **Data Model** is **represented** by **Data Structures**

Files using Data Formats
e.g. YAML, JSON

- Much easier to get started quickly, for exploration
- Decisions about naming attributes, data type, and possible values must be taken
- Helper mechanism for data validation using JSONSchema
- YAML is usually preferred for humans, and JSON for machine consumption, but both are interchangeable

Relational DB
e.g. Nautobot

- Allows relationships between data, but it requires more work to get started, so not well suited for “investigating”
- It’s usually the “final” stage in the network automation journey because of the benefits to data consumption and population

>>> Outcomes of Sketching

First guesses about the **key attributes**

Try to always minimize the data to track!

Identify potential **data grouping**

Device Type, Device Role, Location, etc.
E.g. Maybe all the NTP server IPs are the same for all the devices in each continent?

Expose potential configuration optimization/**simplification**

E.g. Currently the customer has different standards for interfaces, can we consolidate?

Find out data dependencies from **external sources**

E.g. Are the mgmt IPs taken from a pool in an external IPAM service?

Explain your Data Model

Hint: We have a spreadsheet to help you!

>>> How to explain the data organization?

Communicate Internally and Externally about the proposed Data Model design

- At this point, **you should have the following information**:
  - Inventory of all the network devices
  - List of all the features
  - A skeleton data model (in YAML) per feature
  - A good guess about how each feature relates to a device, directly or via grouping
  - Existing sources of information, offline and online
  - Requirements about tooling, and usability of the network automation
- Now, **we need to document, and validate all our assumptions**:
  - Decide where the data model will be stored (tooling), and overall implementation
  - Describe the proposed groupings (levels of intent)
  - List all the features covered, and how they relate to the previous points
  - Offer a detailed description of every data model object

Data Model Spreadsheet, by NTC

Variable Allocation Tab

Use the SoRs, and their features

		Jinja2	Ansible						Nautobot					ServiceNow	HC Vault			
				Group_Vars						Extensibility								
FAMILY	FEATURE		Host Vars	All	Region	Site	Role	...	Core	Config Context	Custom Field	...	Plugin			...	← Storage (examples)	Comment
																	← Details (examples)	
																	Delete Column	
Inventory	Device Name		X															
	Primary IP		X															
	Operating System																	
	OS Version																	
	...																	
Asset	Serial																	
	Chassis																	
	Modules																	
	Interfaces																	
	...																	
Configuration Management	NTP																	
	Local Users																	
	Tacacs/AAA																	
	Login																	
	SNMP																	
	Logging																	
	Banner																	
	DNS																	
	VLAN																	
	Route MAP																	
	Static Routes																	
	Prefix lists																	
	Mac Lists																	
	OSPF									X								
	Connection params																	
	BGP ASN																	
	Timezone																	
	BGP																	
	Services																	
	Spanning-Tree																	
	...																	
...																		

Customize, extend, as needed

Document all the configuration features covered

[Worksheet Source]

>>> Detailed Data Model tab

This will serve as the **Schema Validation** for each model

We define an attribute name, and value type, with its own characteristics, such as where it is stored.

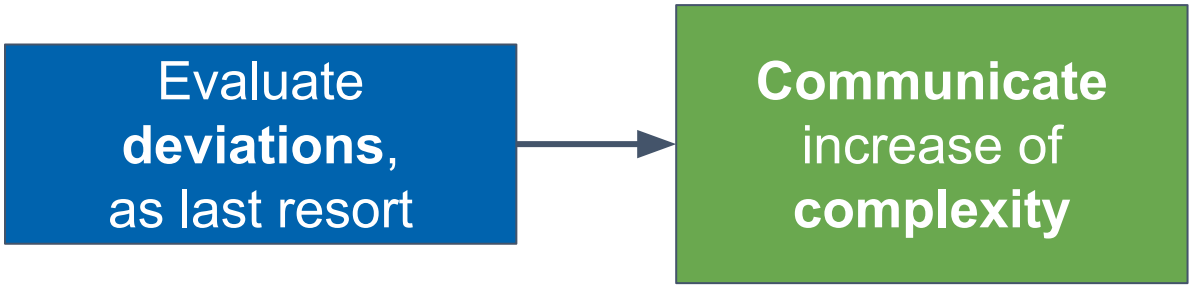
It should help to notice any deviation from the reality, and adapt as needed

Hostname							
Object	**attribute**	**attribute type**	**required (if != True)**	**example value**	**system of record**	**git location**	**description**
	hostname	string			Nautobot		
Logging							
Object	**attribute**	**attribute type**	**required (if != True)**	**example value**	**system of record**	**git location**	**description**
	logging	dictionary			GIT	group_vars	
	logging_enable	boolean					
	logging_servers	list of dictionaries					
	logging_servers_host	IPv4/IPv6		"10.1.1.1" or "FE80::C000:1DFF:FEE0:0"			
	logging_servers_vrf	string		"mgmt"			
	logging_servers_source-interface	string		"Loopback0"			
DNS							
Object	**attribute**	**attribute type**	**required (if != True)**	**example value**	**system of record**	**git location**	**description**
	dns	dictionary			GIT	group_vars	
	dns_enable	boolean					
	dns_servers	list of dictionaries					
	dns_servers_host	IPv4/IPv6		"10.1.1.1" or "FE80::C000:1DFF:FEE0:0"			
	dns_servers_vrf	string		"mgmt"			
	dns_servers_source-interface	string		"Loopback0"			

Always remember that this format is just a proposal that worked fine, but it can/should be improve as needed

>>> Exception Management

This is the golden path, but **does not always work**. Customer will often think they are not cookie cutter, but we have to help them to see the patterns, and move towards configuration standards for a given role



- Additional tech debt
- More complicated data model
- Exception management hooks

Define the Templates

Hint: Templates expand, and transform the Data Model

>>> Templating (with **Jinja2**)

Combine all the data model info, and **expand** using the different levels of intent

Translate the **abstract** data into each specific **interface** model

Templates are also part of the **Source of Truth**

Data from different sources and models could be required for each configuration feature, and it must be combined properly, and expanded depending on the data groupings

All the Data Models must be abstracted from each vendor and interface specifics. The Template will convert this abstract data into each vendor CLI syntax, or interface data model, as needed (taking into account the implicit information in the data model)

Templates are as important as the data model (they work together), so they should be tracked in the proper SoT storage, usually Git

>>> SoT - Configuration template hierarchy

It's a good practice to decompose templates per feature, but as few as possible

Use a hierarchy structure to incorporate inheritance

Usually the top level is used for each platform/interface

```
$ tree templates
.
├── ios
│   ├── catalyst
│   │   ├── 9400
│   │   │   ├── 9406
│   │   │   │   └── vtp.j2
│   │   │   └── main
│   │   │       ├── vtp.j2
│   │   │       └── mgmt_int.j2
│   │   └── main
│   │       ├── vtp.j2
│   │       └── class_map.j2
│   ├── isr
│   │   └── main
│   │       └── services.j2
│   └── main
│       ├── services.j2
│       ├── vtp.j2
│       ├── class_map.j2
│       └── mgmt_int.j2
└── nxos
-- CUT FOR BREVITY--
```

Rare device model override

OS Family override

Most templates are in OS level definition

Jinja2 snippet for composition, using the "os" data variable to select the vendor platform

```
{% set features = ['hostname', 'aaa', 'logging', 'ntp', 'vlan', 'dns'] %}
{% for feature in features %}
{% include os ~ '/' ~ feature ~ '.j2' %}
{% endfor %}
```

>>> Show me my first Jinja2 template!

```
router bgp {{ config_context["bgp"]["asn"] }}
 bgp router-id {{ config_context["bgp"]["rid"] }}
{% if config_context["bgp"]["log-neighbor-changes"] == true %}
 bgp log-neighbor-changes
{% endif %}
{% if config_context["bgp"]["redistribute"]|length > 0 %}
 redistribute {{ config_context["bgp"]["redistribute"]|join(' ') }}
{% endif %}
{% for neighbor in config_context["bgp"]["neighbors"] %}
 neighbor {{ neighbor["ip"] }} remote-as {{ neighbor["remote-asn"] }}
{% endfor %}
```

Do not take the conclusion that only CLI rendering is possible, choose your interface model

```
<config>
   <native xmlns="urn:ios">
      <interface>
         <{{ base }}>
            <name>{{ interface_id }}.{{ vlan_id }}</name>
            <encapsulation>
               <dot1Q>
                  <vlan-id>{{ vlan_id }}</vlan-id>
               </dot1Q>
            </encapsulation>
            <ip>
               <address>
                  <primary>
                     <address>{{ ip }}</address>
                     <mask>{{ subnet_mask }}</mask>
                  </primary>
               </address>
            </ip>
         </{{ base }}>
      </interface>
   </native>
</config>
```

A Template **assumes** some data model to reference as variables, so the Data Model representation, and the Template expectations **must** match

More examples in: ⟶ https://github.com/nautobot/demo-gc-templates

>>> Data Structure Mapping Example - Advanced

```yaml
---
bgp:
 router_id: "10.1.1.1"
 asn: 121
 neighbors:
   - name: "Verizon"
     asn: "111"
     ip_address: "1.1.12.1"
     prefix_in: "filter_in"
     prefix_out: "filter_out"
     route_map_in: "filter_by_community"
     route_map_out: "set_as_prepend"
     community: true
     soft: true
   - name: "ATT"
     asn: "131"
     ip_address: "1.1.23.3"
   - name: "BACKUP iBGP Router"
     asn: "121"
     ip_address: "10.0.0.2"
 networks:
     - "10.0.0.0/24"
     - "10.0.1.0/24"
     - "10.0.2.0/24"
     - "10.0.4.0/24"
```

YAML

```
router bgp {{ bgp["asn"] }}
 no synchronization
 bgp router-id {{ bgp["router_id"] }}
 bgp log-neighbor-changes
{% for network in bgp["networks"] |
ipaddr('host/prefix') %}
 network {{ network | ipaddr('network') }} mask {{
network | ipaddr('netmask') }}
{% enfor %}
 timers bgp 5 15
{% for neighbor in bgp["neighbors"] %}
 neighbor {{ neighbor["ip_address"] }} remote-as {{
neighbor["asn"] }}
 neighbor {{ neighbor["ip_address"] }} description {{
neighbor["name"] }}
{% if "community" in neighbor and
neighbor["community"] %}
 neighbor {{ neighbor["ip_address"] }} send-community
{% endif %}
 # omitted for brevity
{% enfor %}
 no auto-summary
```

Jinja

```
router bgp 121
 no synchronization
 bgp router-id 10.1.1.1
 bgp log-neighbor-changes
 network 10.0.0.0 mask 255.255.255.0
 network 10.0.1.0 mask 255.255.255.0
 network 10.0.2.0 mask 255.255.255.0
 network 10.0.3.0 mask 255.255.255.0
 timers bgp 5 15
 neighbor 1.1.12.1 remote-as 111
 neighbor 1.1.12.1 description Verizon
 neighbor 1.1.12.1 send-community
 neighbor 1.1.12.1 soft-reconfiguration inbound
 neighbor 1.1.12.1 prefix-list filter_in in
 Neighbor 1.1.12.1 prefix-list filter_out out
 neighbor 1.1.12.1 route-map filter_by_community in
 neighbor 1.1.12.1 route-map set_as_prepend out
 neighbor 1.1.12.3 remote-as 131
 neighbor 1.1.12.3 description ATT
 neighbor 10.0.0.2 remote-as 121
 neighbor 10.0.0.2 description BACKUP iBGP Router
 no auto-summary
```

Config


Show Compliance

>>> Show Compliance

You should **demonstrate** that it is possible to replicate the target configuration using the data model and the templates

Reference Configuration

Rendered Configuration

Data Model

Template

*Nautobot **Golden Config plugin**, or an **Ansible Playbook** are common ways to validate this compliance*

Check Customer’s point

>>> Customer's Point

- The Customer may not be fully aware of what/how the Data Modeling will help him
- After the first demo of Configuration Compliance, he would provide new feedback that could help us to refine the proposal:
  - Uncover missing System of Records
  - Explain some tribal-knowledge not taken into account
  - Agree on, or propose, some configuration optimization to simplify design
  - Features prioritization
  - New use-cases for the Data Model
- This is a great opportunity to get the feedback that will **trigger again the Data Modeling loop** to extend, or refine, the proposal

Lessons learned

>>> Most Common Mistakes

Expecting the customer to define the data model for you, rather than guiding the customer and checking in with your proposal

Following the vendor configuration 1-to-1 to the data model, rather than intention of the config

Following a vendor model, rather than abstractly thinking "would this also work for JUNOS"

Not considering the characteristics of the data, and where it should live

Making the data model too complex to be usable by customers

Mapping the running state from the network without abstraction. The intended state will generate it, but does not need to have the same representation

Managing data to manage data, rather than finding ways to make the amount of data managed minimal

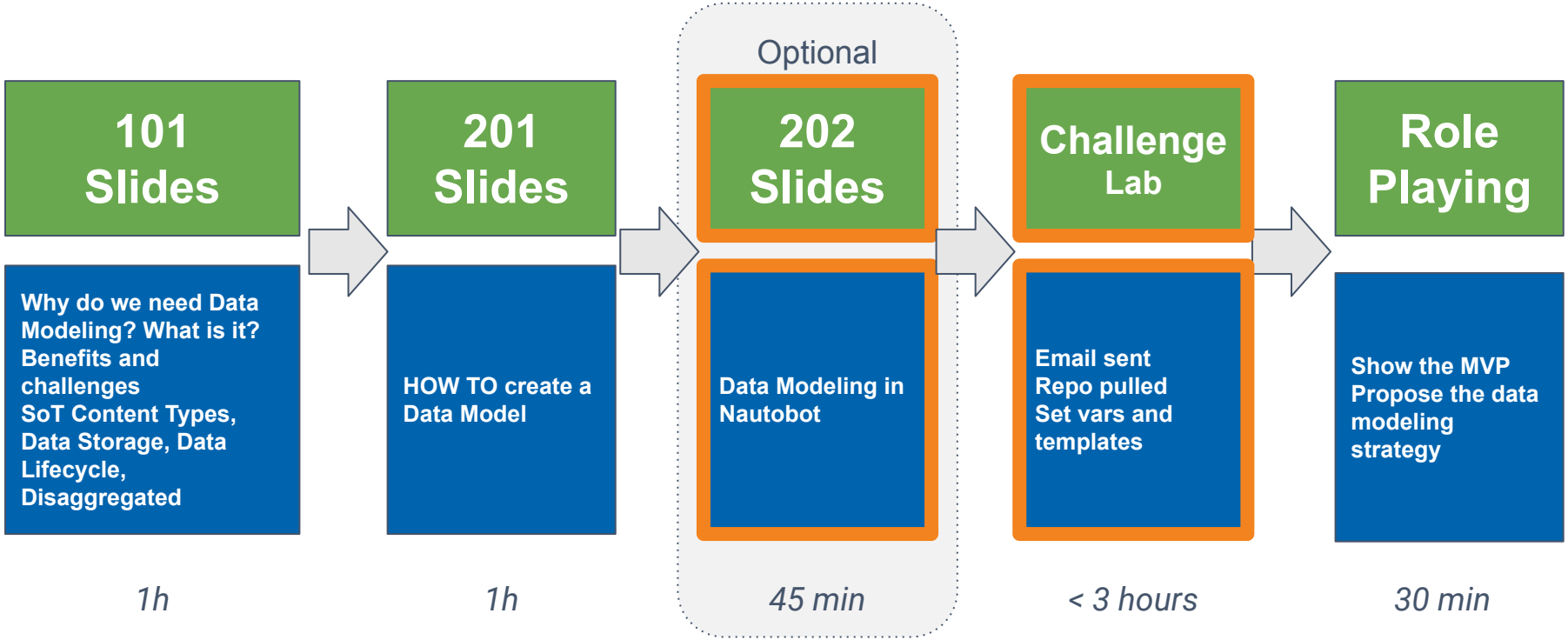
e.g. managing bgp timers, when they are always "5 15" or managing console timeout, when always the same

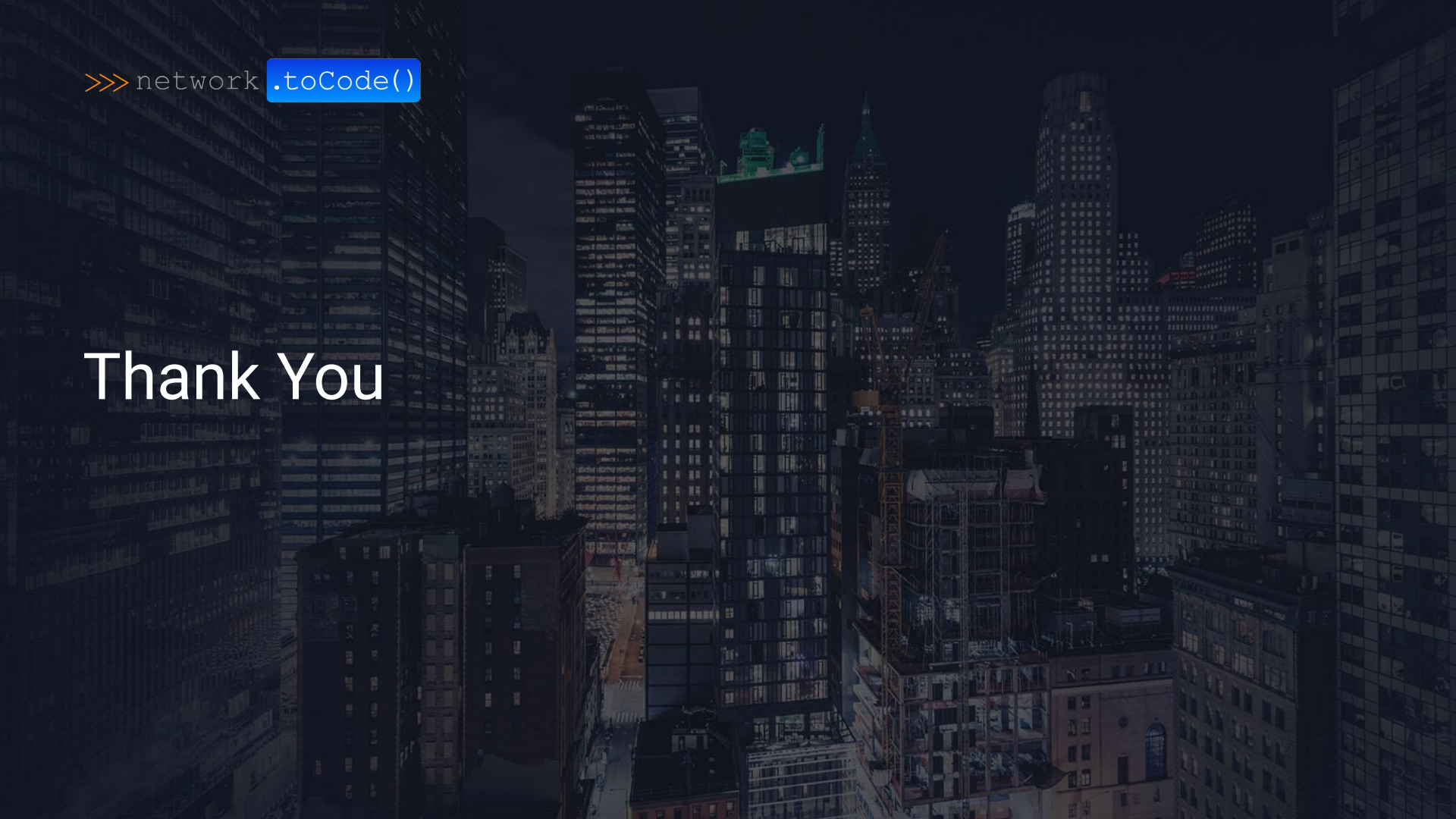
Attempting to have multiple conflicting sets of hierarchy, such as device type and regional

You must choose one clear hierarchy, such as having {region} | {cluster} | {site}, and then providing if conditionals for hardware

Next Step: Data Modeling Challenge
Optional: 202 Data Modeling in Nautobot

Data Modeling Training Plan




>>> network .toCode()
Thank You