

Crafting network configuration at Meta

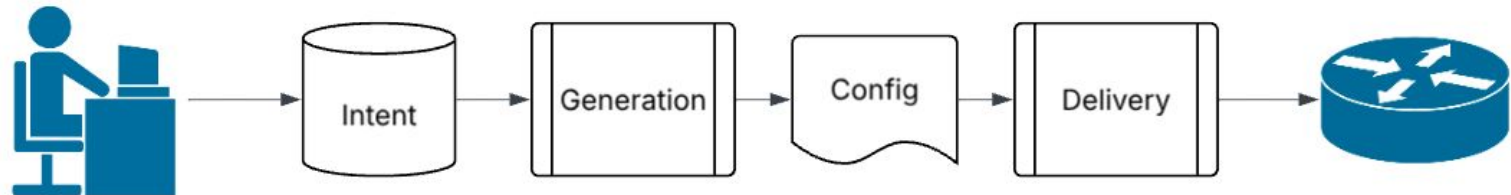
03-Jun-2026



Santiago Alfonso
Muñoz Rodríguez

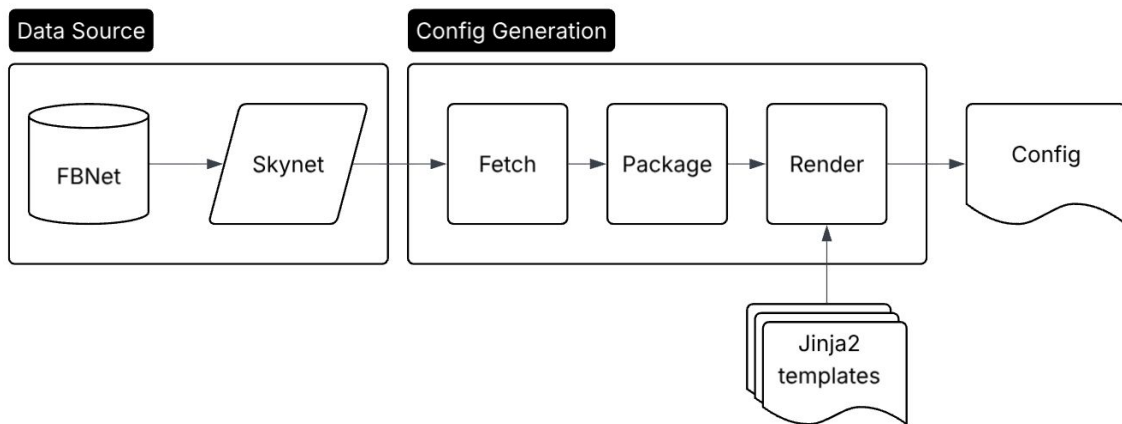
High-level picture

- Network is **defined** and stored in a database
 - This is an **abstract schema** that represents a blueprint
- This intent is then is **translated** into device configurations
 - Using **vendor schema** and protocols (CLI/Netconf...)



Three moving pieces

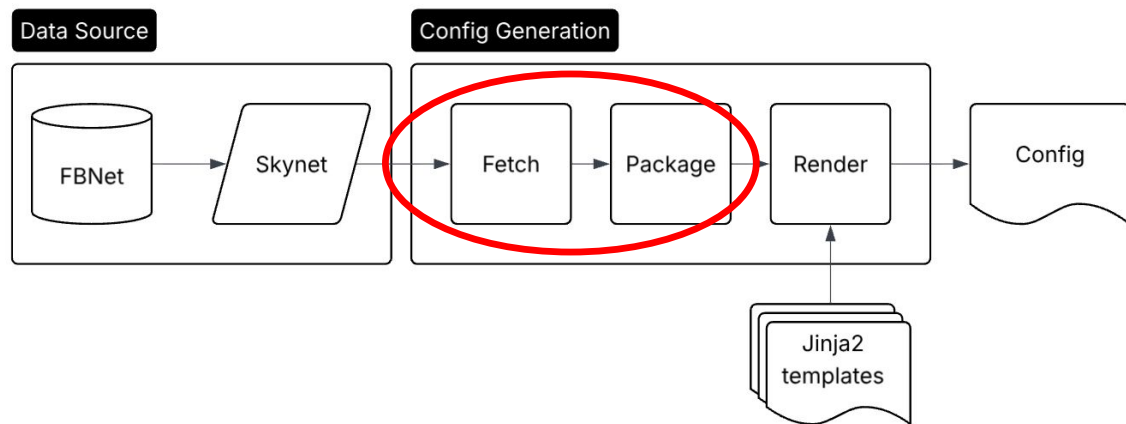
- **FBNet**: Object store. Every network element is an object
- **Skynet**: Thrift query API. Exposes data in key-value form
- **Config Gen**: Uses Skynet + Jinja2 to craft the configs



Jinja and data pipelines

Jinja requires data to be exposed to the template

Building and maintaining this pipeline is costly



Skynet + X-path = Skypath

A **query DSL** with minimalistic syntax inspired by XML

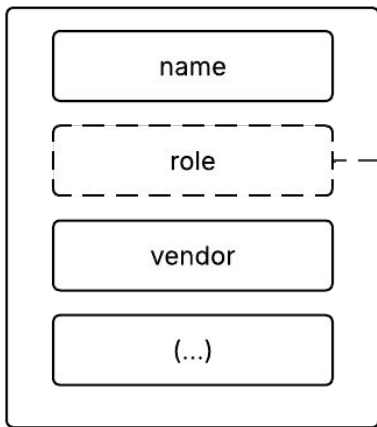
```
DesiredBackboneRouter[name={ }]/role/name
```

Table

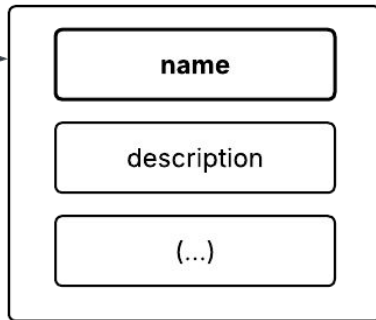
Filters

Predicate

DesiredBackboneRouter



DesiredNetworkRole



Some example queries

```
BgpPrefixLimit[peertype={}] [peer_as={}]/value
```

```
PhysicalIface[device={}] [name={}]/slot/speed_mode
```

```
OpticalChannel[id={}]/a_interface/name
```

Application to Jinja templates

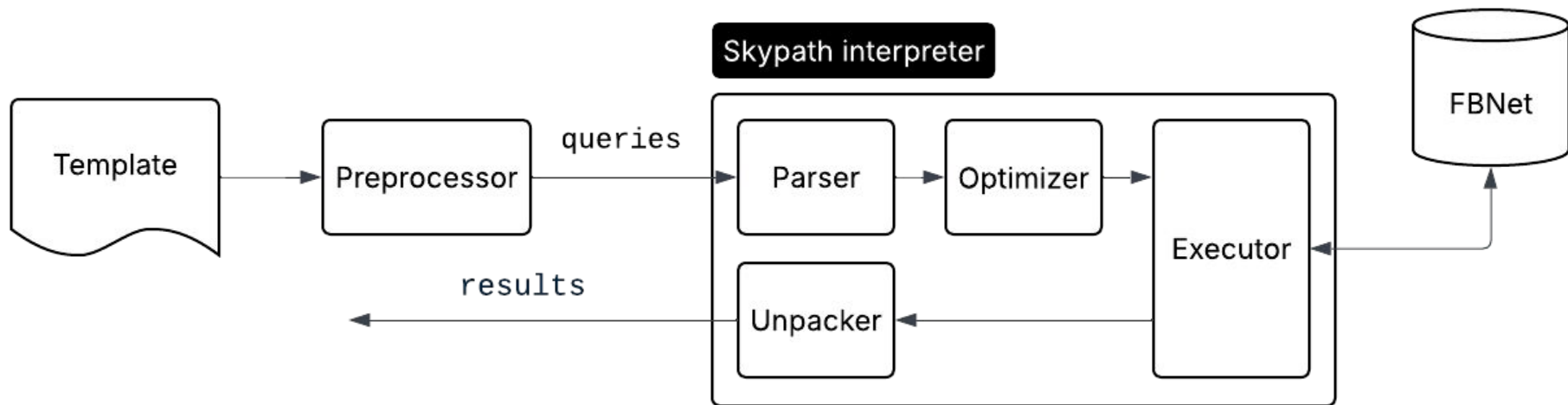
```
{# for a specific device and interface #}  
router bgp {$ Router[name={dev}]/asn $}  
  neighbor {$ Circuit[interface={ifc}]/other_side/v6addr $} route-map NOTHING in  
  neighbor {$ Circuit[interface={ifc}]/other_side/v6addr $} route-map NOTHING out
```

The template becomes a specification that enables
self-contained reasoning

Authoring stay close to *writing a plain config*

Query execution

Template is preprocessed to allow optimized querying
The interpreter acts as a **lazy evaluator**



The result

```
{# for a specific device and interface #}  
router bgp {$ Router[name={dev}]/asn $}  
  neighbor {$ Circuit[interface={ifc}]/other_side/v6addr $} route-map NOTHING in  
  neighbor {$ Circuit[interface={ifc}]/other_side/v6addr $} route-map NOTHING out
```



```
router bgp 123456  
  neighbor 2001:db8:1:1::2 route-map NOTHING in  
  neighbor 2001:db8:1:1::2 route-map NOTHING out
```

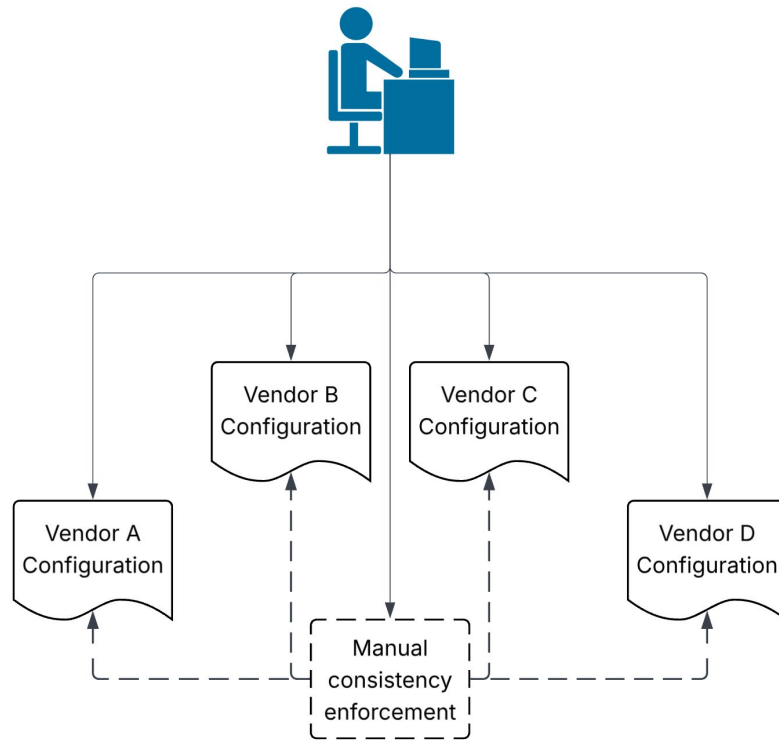
One step further: Multi-vendor

03-Jun-2026

The multi-vendor challenge

North star

To be able to operate a multi-vendor network as if it was still single-vendor

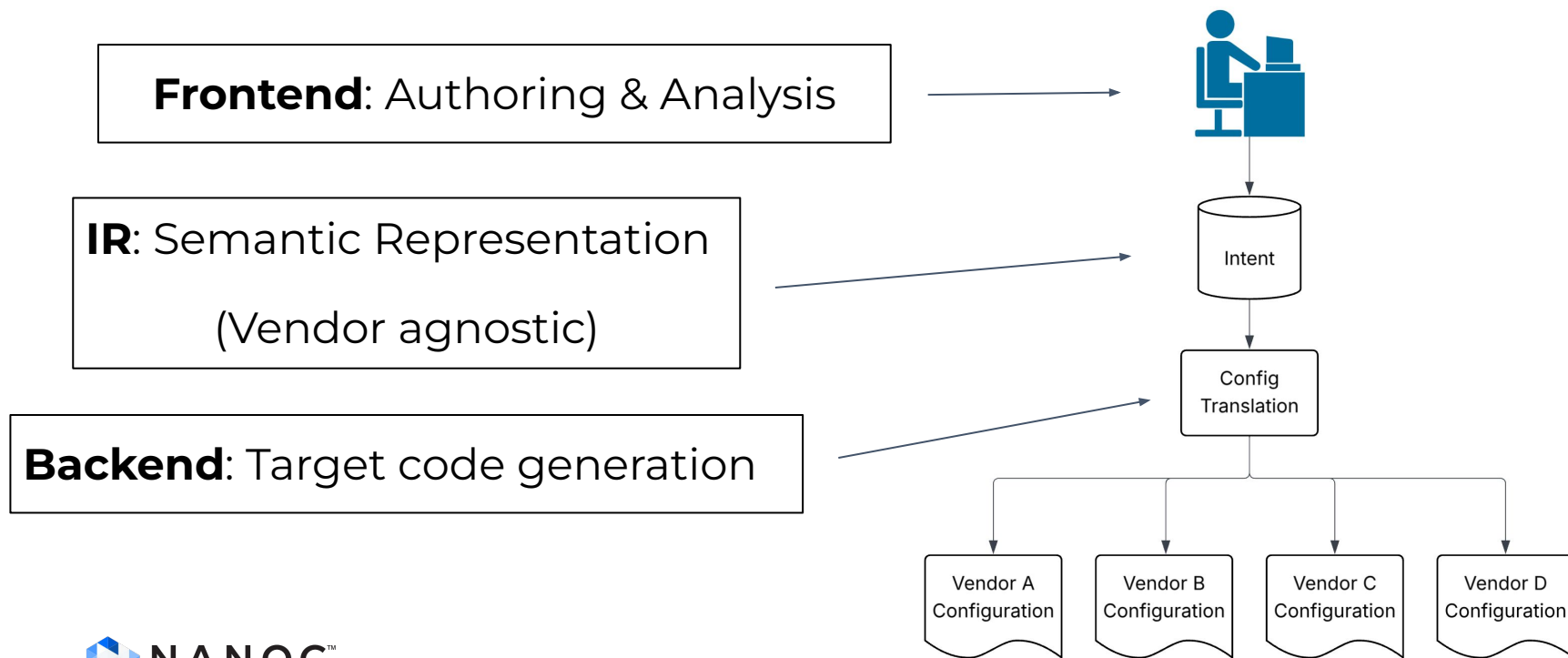


Inspiration from compilers

One typically does not need to write a computer program one time for each CPU architecture (x86, arm, mips...)

Hypothesis: Configurations are like computer programs

A compiler for configs = Confpiler



Frontend: High-level representation

- Favor friendly means of authoring intent
 - The closer to CLI the better
- IR schema must be vendor agnostic
 - OpenConfig is a good example

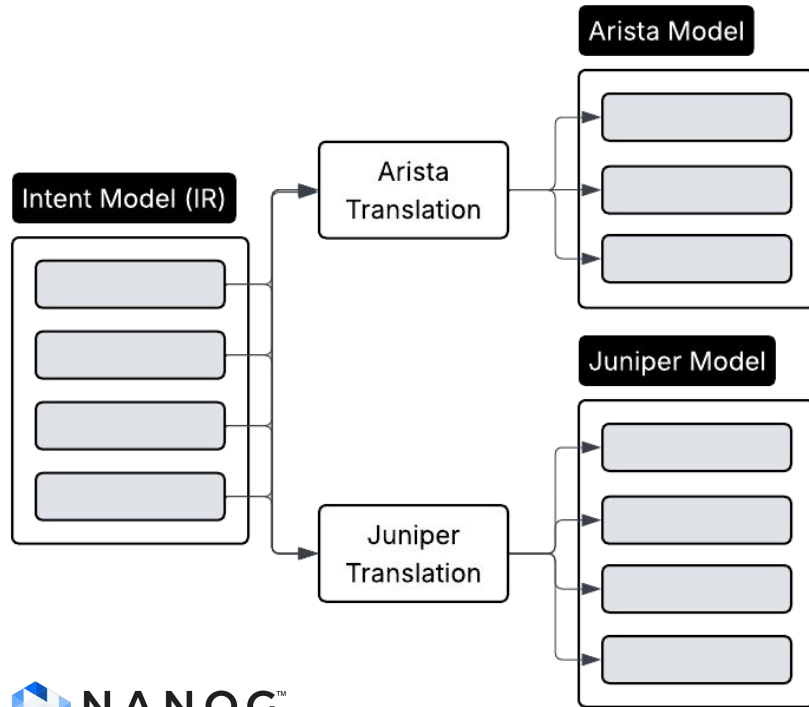
CLI-like authoring

```
1 statements {  
2   ROLEA <- ROLEA {  
3     1 permit ROLEA-ROLEA-AGG  
4   }  
5   ROLEA -> ROLEA {  
6     1 permit ROLEA-ROLEA-AGG  
7   }  
8 }
```

Front-end
Compilation

Intent Model (IR)

Backend: Schema conversion (Yang)



Step one

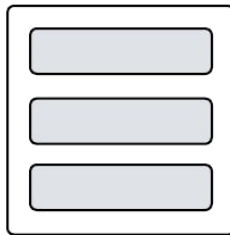
Translate from IR schema to
vendor-schema(s)

Backend: Serialization

Step two

Serialize to the desired
delivery protocol

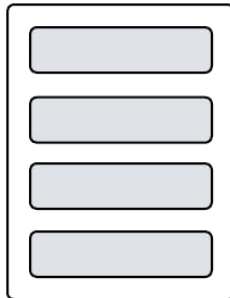
Arista Model



Arista
Serialization

```
1 interface Ethernet1
2   description a sample interface
3   ip address 10.0.0.1/24
4   no shutdown
```

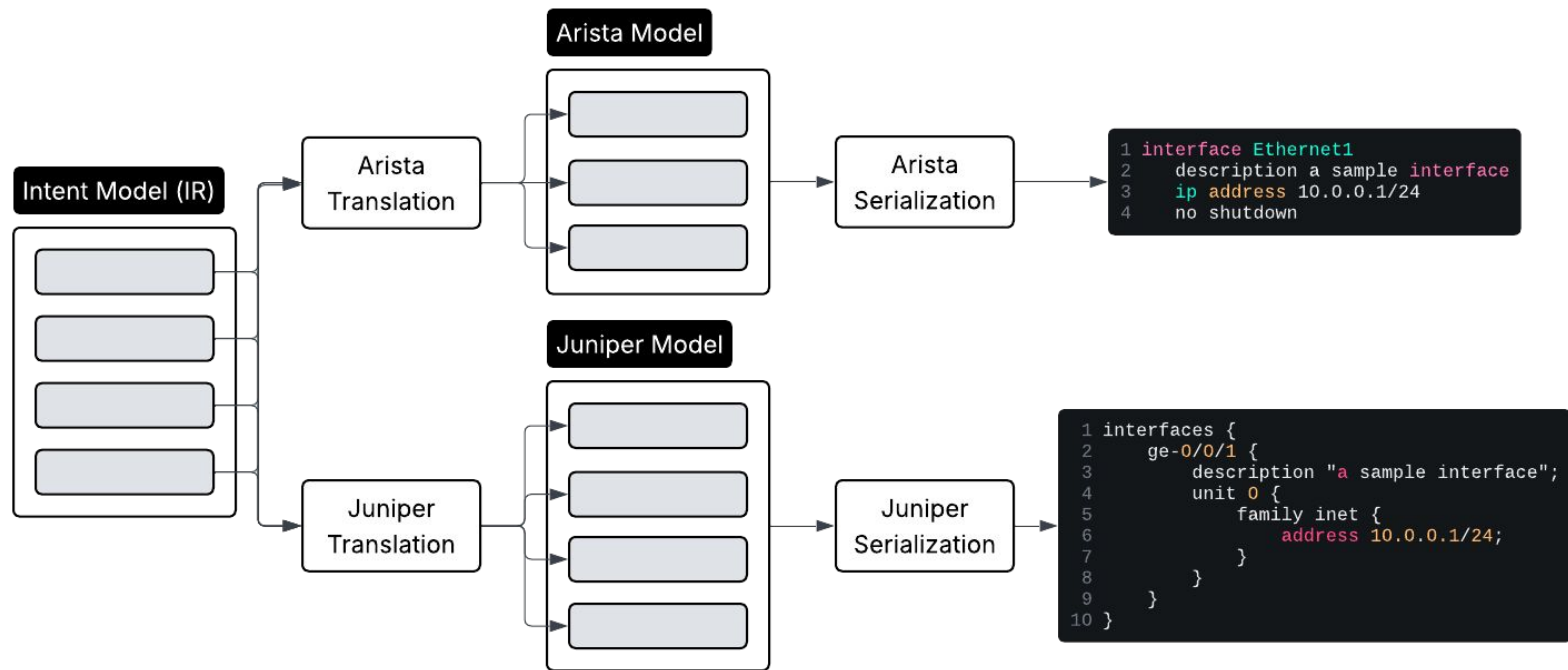
Juniper Model



Juniper
Serialization

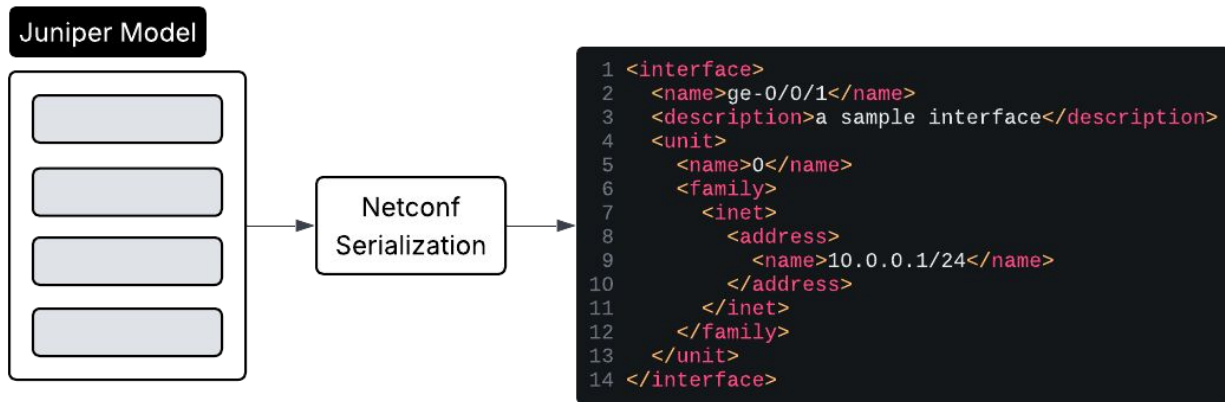
```
1 interfaces {
2   ge-0/0/1 {
3     description "a sample interface";
4     unit 0 {
5       family inet {
6         address 10.0.0.1/24;
7       }
8     }
9   }
10 }
```

Backend: The complete picture



Property: Protocol agnosticism

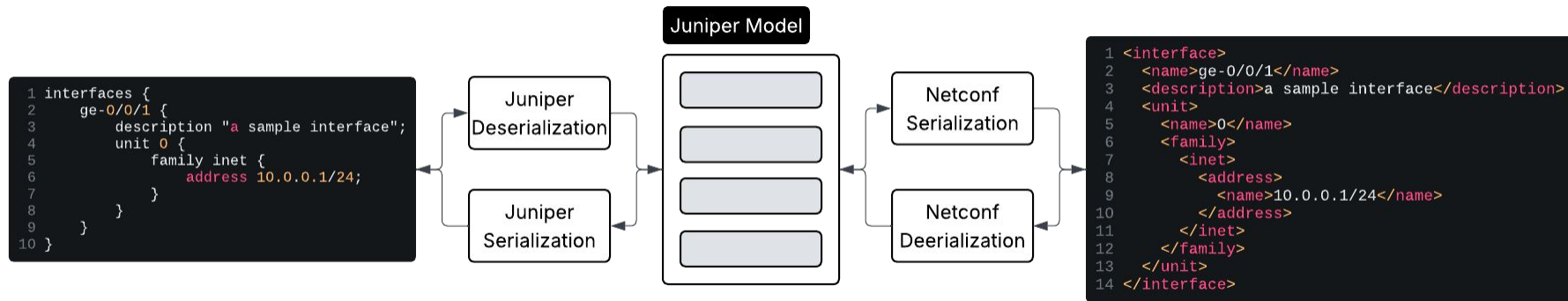
Separating schema translation from serialization allows
switching delivery protocols on the fly



Property: Protocol translation

Humans work with human languages: CLIs

Machines work with machine languages: Netconf/GNMI



The output: Jinja2 with Skypath

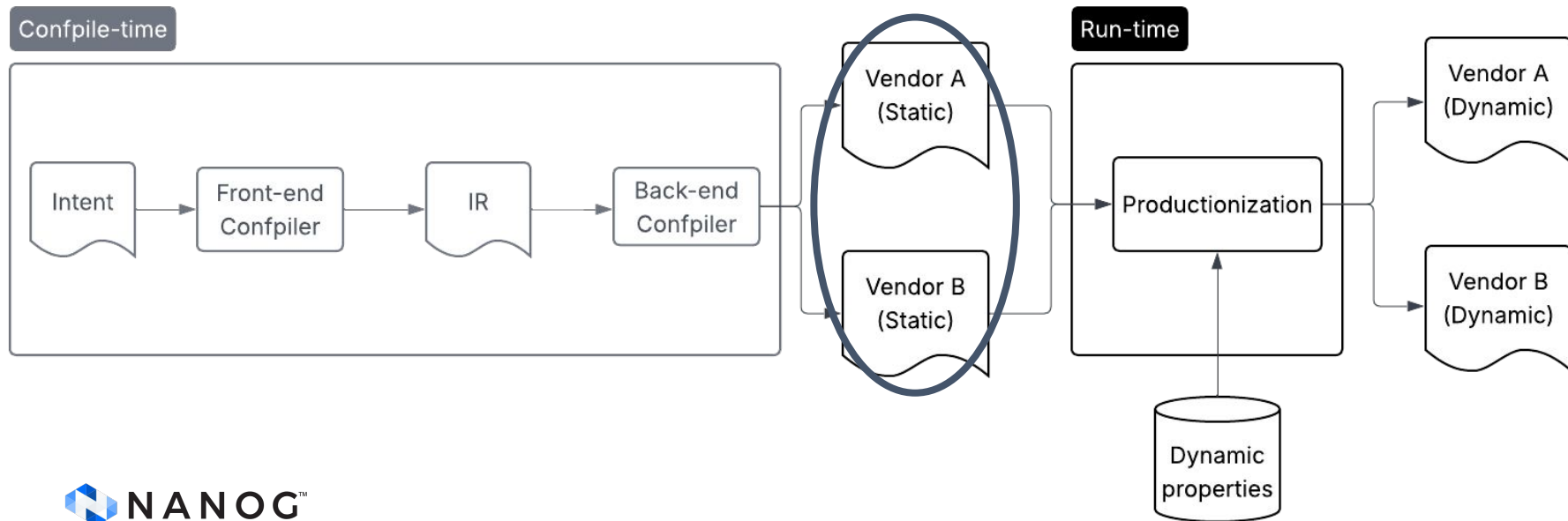
```
ip prefix-list LOOPBACK-V4  
  seq 10 permit {$ LoopbackInterface[dev={}]/v4addr $}  
  seq 20 deny 0.0.0.0/0 le 32
```

Serves as **drop-in replacement** for existing pipelines

Skypath is used for values not known at “*confpile*” time

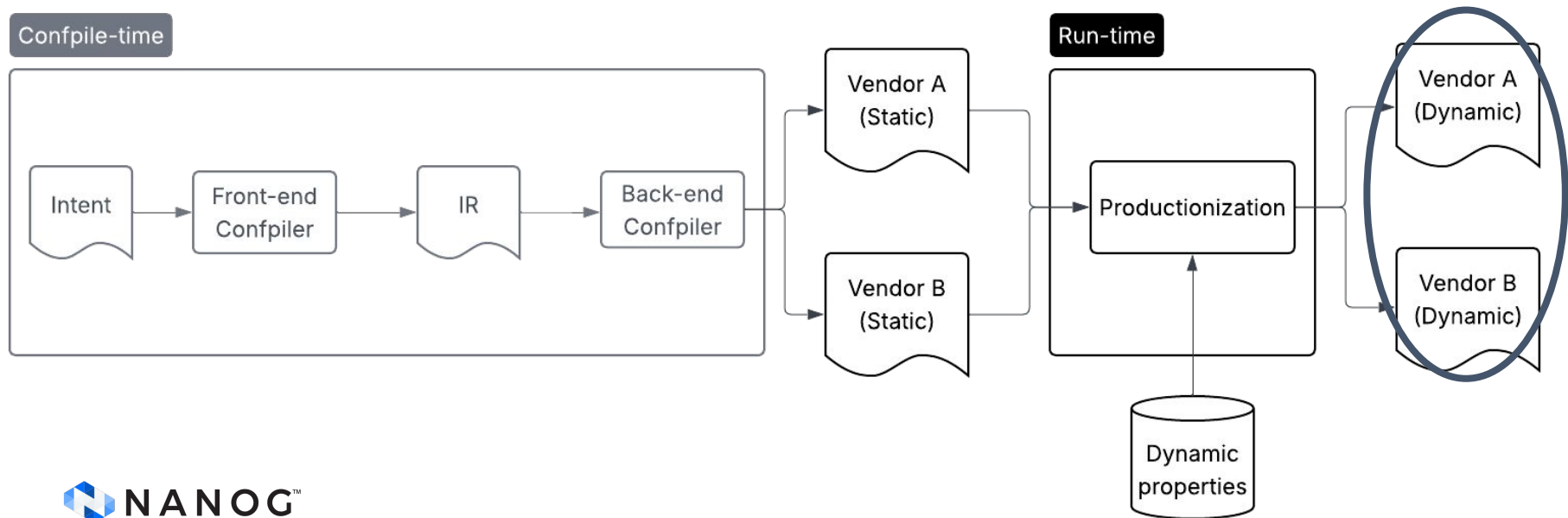
Compile-time: static blueprint

Like Java **bytecode**. Some values are known, some are not.



Run-time: dynamic configurations

Rendering is like running a program with different inputs



Lessons learnt along the way

Modelling: Our ability to be *declarative* can only as good as our database models. Simple representations will not fix your models

User experience: There is a limited budget for change, respect the established tools of the trade

Interoperability: Abstraction only makes sense when behaviour is portable. Not everything in this world is portable

Summary

A DSL allows us to build self-contained specifications, which come with interesting cognitive gains

Portable parts of configuration and intent can be abstracted with a compiler-like construct

Schema/protocol separation allows us to use the right tool for each type of task



Thank you

03-Jun-2026