GUIDE FOR NETWORK ENGINEERS

Services and templates: view, test and extend without development

How to see what each service generates on your devices, test a change before saving it, and add what the template doesn't include. You work with parameters; Oktavia takes care of the template, the gate and the tests.

Built by Oktanet | Mexico · 2026 | Public document

CONTENTS

1. Who this guide is for
2. The idea: parameters, not Jinja
3. Your domain's services
4. The service page
5. Declared values and "See what changes"
6. Test with a device
7. Explain this service
8. IOS XR: no Jinja, XML instead
9. NETCONF dictionary: search like the CLI
10. Service Builder
11. Extending a service
12. The gate before publishing
13. An extension's network tests
14. Custom services
15. With Argos: Services by domain
16. Recipes
17. How templates and values work
18. With Ansible and the API
19. Good practices

1. Who this guide is for

For network and automation engineers, at customers or partners, who maintain a network's services in Oktavia: Campus, WAN, Security and Management in an enterprise; Transport, Services, Security and Management at a provider.

By the end you will be able to read a service end to end, test a value against a real device without saving anything, and add to a service what its template doesn't include, without writing Jinja and without asking for development.

This guide complements the Intent guide, which explains scopes and the change cycle. If this is your first time with Oktavia, start there.

2. The idea: parameters, not Jinja

Every Oktavia service has a tested base template that turns values into configuration. That template **is not edited**: it is **extended**. That way an Oktavia update doesn't overwrite your work, and your work doesn't break the template.

Your work as an engineer is in the **parameters**: which values, in which scope and with which network test. An extension's template comes from the Oktavia catalog or is proposed by Argos, and a deterministic gate decides whether it can be published.

PIECE	WHO MAKES IT	WHERE IT LIVES
Base template	Oktavia.	The service catalog. Not touched.
Extension	The Oktavia catalog, with one click, or Argos from your description.	Next to the service it extends.
Values	You.	The intent (`service_vars`), in the scope you choose.
Tests	Oktavia derives them from the template and the parameters.	The compliance of every audit.

Saving values does not touch the network. They reach the devices through the usual cycle: audit, compliance, Digital Twin, you approve, and verification confirms it.

3. Your domain's services

Open **Services** and pick the domain. The tab formerly called Intent is now called **Services**: it lists every service in the domain, declared or not.

COLUMN	WHAT IT SAYS
Platform	Which systems the service exists for: `ios`, `iosxe`, `xr`…
Declared in	The scopes where it has values.
Devices	How many devices it applies to.
Failures and %	Its compliance in the last audit.
Tests	How many config and state tests validate it; "no tests" if none.

Undeclared services come last, with a link to declare them. The domain **Summary** also shows each service's platform.

4. The service page

Each service in the list opens its own page. It gathers in one place everything you need to understand and change it:

1. **Declared values**: every scope where it is declared, with its values, and the buttons to edit them,

view the template and extend (section 5).

2. **Test with a device**: values, template and result, side by side, for a real device (section 6).
3. ✦ **Explain this service**: Argos explains what it does and what it generates (section 7).
4. ＋ **Add parameters** (Change a service): what you can add to it (section 11).
5. **Devices where it is active**, by site, with ⚠ on the failing ones, and **Tests that validate it**.

Tests come in two kinds. **Config** tests compare the running configuration with what the template generates, one rule per device. **State** tests check that the service works on the network.

5. Declared values and "See what changes"

The **Declared values** fold shows every scope where the service lives —`defaults`, `by_role`, `by_site`, `by_device`, `by_model`, `by_platform`…— with its values. **Edit values**, **Template** and **Extend here** sit on the same row.

Edit and see what changes

1. Click **Edit values** on the scope you want to change. A JSON editor opens.
2. Change the values.
3. Click **See what changes**. Oktavia shows, per device, the lines of the desired configuration that would change: – what is there today, + what your change leaves. Secrets stay hidden.
4. If it is what you expected, click **Save**. If not, adjust and look again.

```
r1
  router bgp 65001
+  neighbor 198.51.100.1 timers 10 30
```

These are real configuration lines, not just values. If a device you didn't expect shows up in the list, your scope is wider than you thought: don't save yet.

> **What you need to know.** Editing requires being an organization administrator. Secrets always go as a vault reference, `<<vault:…>>`; never in clear text.

6. Test with a device

This is the best way to understand a service. Pick a device from the menu, grouped by site, and you get three columns side by side:

COLUMN	WHAT IT SHOWS
1. Values	The device's values for this service and which scopes they come from. Each scope is a button that opens its declaration for editing.
2. Template	The service's Jinja template.
3. What it generates	The configuration it generates today on that device.

The template, the JSON and the XML are colored as in VS Code. In column 3, whatever comes from the device's parameters is **highlighted**; hover over it to see which parameter it comes from. Anything not highlighted is fixed in the template.

Test these values

In column 1 you can change the values and click **Test these values**. Oktavia renders with the same engine used by audits and shows you the difference. **Nothing is saved.** Use it to answer "what would happen if…?" before touching intent.

> **If the test says "nothing changes".** Some values come from the device inventory, for example `ospf_config.router_id`, and win over the service's values. If you change one of those in the service, the configuration doesn't move. Check the device inventory.

7. Explain this service

Click ✦ **Explain this service** and Argos, with local AI or with Claude, explains:

- what the service does;
- each parameter;
- what it generates on the device you picked;
- tips for using it well.

Argos receives the template, the masked values and the generated output. It never receives secrets or the device's full running configuration.

8. IOS XR: no Jinja, XML instead

On IOS XR there is no Jinja template. The NETCONF lane builds the YANG XML from the same values. The service page works the same way, with these differences:

- **Column 2**: explains how the NETCONF lane builds the configuration.
- **Column 3**: shows the XML for the chosen device. The device's values are highlighted here too, so you can tell what comes from variables from what is fixed in the lane.
- **See what changes** and **Test these values** compare today's NETCONF plan with the plan for your change, exactly as the Digital Twin would push it, with *commit confirmed*.

This applies to core services and to customer services. There is a recipe in section 16.

9. NETCONF dictionary: search like the CLI

You do not need to know NETCONF to extend an IOS XR service. **Services → NETCONF dictionary** holds the parameters the device accepts, read from its own YANG models, with type, range and default.

1. **Search like the CLI**: "neighbor timers keepalive", "isis lsp-gen-interval", "interface mtu". You can also paste an XR line such as `neighbor 192.0.2.1 timers 10 30`: Oktavia ignores the values and finds the leaves.
2. **Pick the parameter.** Each result is shown like the CLI —"router bgp ‹as› › neighbor ‹ip› › timers › keepalive-interval"— and the variants (VRF, session-group…) are grouped.
3. **Use it in a service.** Oktavia builds the extension with no AI: the XML with its context (the AS comes from core BGP and the IS-IS process), the field with its range and the removal. You only enter the value, with the preview and "Save and propose".

If you already have the XML you want on the device, **From XML** turns it into parameters: every leaf with text becomes a field, repeated siblings become a list and anything that looks like a secret stays as `<<vault:… >>`.

10. Service Builder: build the service from the device's own options

Services → Service Builder brings into one screen what used to take several steps: you pick the options the device accepts, Oktavia builds the template and the form, you see the result as IOS XR CLI and as NETCONF XML, and you publish it with its preview. You don't write YANG models or service packages: the options come from the device's own YANG (the same index as the NETCONF dictionary).

IOS XR · NETCONF path, step by step

Example: BGP keepalive and hold per neighbor on the PEs.

1. **Search like the CLI**: "neighbor timers keepalive", or paste an XR line such as `neighbor 192.0.2.1 timers 10 30`. Tick *keepalive-interval* and *holdtime*; they show as "router bgp ‹as› › neighbor ‹ip› › timers".
2. **Pick the base service and the scope**: "Core BGP (IOS XR)" on the `pe` role. The AS comes from the base service; you don't type it again.
3. **Fill in the form.** Each option is a field with its range (0..65535) and the neighbor list comes pre-filled with the scope's real neighbors; you only enter 10 and 30.
4. **Look at the result** in the tabs on the right: *XR CLI*, *NETCONF XML* and *Translate*.
5. **Save and propose the change.** It goes through the gate (syntax, secrets, impact on other services), shows what changes per device and proposes the change in the Digital Twin: the dry

run runs *validate* on the devices and you approve and confirm with confirmed commit.

```
router bgp 64500
 neighbor 192.0.2.1
  timers 10 30
 neighbor 198.51.100.1
  timers 10 30
```

```
<router xmlns="http://cisco.com/ns/yang/Cisco-IOS-XR-um-router-bgp-cfg">
  <bgp><as><as-number>64500</as-number>
    <neighbors>
      <neighbor><neighbor-address>192.0.2.1</neighbor-address>
        <timers><keepalive-interval>10</keepalive-interval><holdtime>30</holdtime></timers>
      </neighbor>
    </neighbors>
  </as></bgp>
</router>
```

Translate CLI ⇄ XML

The *Translate* tab turns IOS XR CLI into the NETCONF XML that represents it, and back. The translation is **deterministic**: it comes from the device's YANG (the `Cisco-IOS-XR-um-*` models), not from an AI. With the lab's real configurations, XML → CLI → XML comes back identical for IS-IS, BGP, L2VPN, QoS, Segment Routing and telemetry. Whatever the dictionary doesn't know is flagged, never invented. *Coming soon:* "live" translation, where the device itself converts the CLI without committing and discards the result.

IOS / IOS-XE · CLI path

1. **Describe what you want** ("IP SLA towards the provider on the WAN routers") and, if you have it, paste the config of a device that already has it. Lines with secrets are removed before anything reaches the AI.
2. **Argos drafts the Jinja template** (local AI or Claude) with its parameters.
3. Same form, same per-device preview and the same **Save and propose** button.

The previous editor is still available as the *Advanced editor* for anyone who wants to write the template by hand.

11. Extending a service

When the base template doesn't include what you need, you extend it. There are two ways in:

- **Advanced options**: tested extensions from the Oktavia catalog, enabled with one click. For example, in OSPF: timers, BFD and area types; in BGP: per-neighbor timers, address-family and aggregates; on interfaces: `bandwidth`.
- **＋ Add parameters**, on the row of the scope where you want the values, or from "Test with a device".

Change a service: one question, one form and one button

1. **What do you want to change?** Say it in your own words, for example "keepalive 10 and hold 30 for the RR neighbors". If it matches an extension in the Oktavia catalog it is picked automatically, no AI; otherwise Argos generates it (local AI or Claude). The catalog shortcuts stay below as buttons.
2. **Form.** The fields come from the extension, with label, range and default; lists are tables with "+ row", and it comes pre-filled with what is real in the scope, such as your neighbors. JSON and template stay under "Advanced".
3. **Automatic preview.** When you change a valid value you see, per device, the lines that change (on IOS XR, the XML it adds). If the gate rejects something, it says so in one line.
4. **Save and propose the change.** One button: it publishes the extension, leaves the values in the scope and proposes the change in the Digital Twin. It is enabled only when the latest preview passed and matches the current values.

The same experience is in Argos: **Custom orchestration → Change a service**, or type "change a service".

The extension's values stay in the intent (`service_vars`), in the same scope where you clicked **+ Add parameters**. They reach the devices through the cycle: audit → compliance → Digital Twin → you approve → verification.

12. The gate before publishing

Before an extension is published, a deterministic gate checks it. It is not an AI opinion: these are fixed rules.

RULE	WHAT IT CHECKS
Blocked commands	The template carries no forbidden commands.
No literal secrets	Not even type 7. Secrets go as `<<vault:…>>`.
Clean removal	Removing the extension only uses `no` or `default`.
Collateral	If the extension changes lines that belong to another service, it is not published.

The preview renders the extension on real devices without touching the network, and it is signed: **only what passed the gate with those same values is published**.

13. An extension's network tests

An extension doesn't end at the configuration: it also says how to check that it works. There are two kinds of test.

State tests

They check that something is up: BGP or OSPF neighbors, interfaces, BFD, IS-IS, routes. They are derived from the parameters. For example, `neighbors[].address` yields the list of neighbors that must be established.

Value tests

They compare what is seen on the network with what you declared. It is not enough for the neighbor to be up: it also has to have the right value.

- BGP timers negotiated with each neighbor.
- OSPF hello and dead per interface.
- SNMPv3 users: group and protocols, without keys.

The catalog extensions `bgp_timers` and `ospf_timers` already ship with their value test. When you audit you will see, for example, "BGP · timers negotiated with the neighbor". Value tests exist today on IOS and IOS XE.

14. Custom services

When what you need doesn't extend an existing service, create a new one in **Services → Custom services**.

1. Describe the service or paste a sample configuration.
2. Argos proposes the template, the fields and a sample of values.
3. The same gate from section 12 decides whether it can be published.

From there you can also open **View template** for any service, to read it without editing it.

15. With Argos: Services by domain

If you'd rather ask in your own words, open **Argos → Configure → Services by domain**. What used to be the Configurator assistant now lives here.

1. Pick the domain and the service. Only those for the platforms in your network appear.
2. Say which values you want and where, for example `timers 10 30 en los wan_router`. You can also bring them in from a device backup.
3. Argos proposes the change. Click 🔍 **What changes on the devices** to see the desired configuration per device before saving.
4. If it is what you wanted, click **Apply and close the cycle**. You still approve the change in the Digital Twin.

16. Recipes

BGP · Timers 10/30 on one neighbor

You want to detect a provider failure sooner. The BGP template has no per-neighbor timers; the catalog extension `bgp_timers` does.

1. In **Services → WAN**, open the BGP service.
2. In **Declared values**, find the router's scope, usually `by_device`, and click **+ Add parameters**.
3. Pick **BGP · per-neighbor timers** from the catalog and fill in the parameters:

```
{
  "neighbors": [
    { "address": "198.51.100.1", "keepalive": 10, "hold": 30 }
  ]
}
```

4. Click **See what changes**. A single new line should appear, on that device:

```
r1
  router bgp 65001
+  neighbor 198.51.100.1 timers 10 30
```

5. Save and close the cycle. Approve the change in the Digital Twin.
6. After verification, review the device's compliance: the value test **BGP · timers negotiated with the neighbor** appears. The neighbor negotiates the lower hold: if the other side has a lower one, that is what you will see.

Interfaces · Add `bandwidth`

The interfaces template has no `bandwidth`, and you need it for OSPF or EIGRP cost and for a percentage-based QoS. `bandwidth` does not change the real link speed.

1. Open the router's interfaces service and, in **Advanced options**, enable **Interfaces · bandwidth** (`interface_bandwidth`).
2. Fill in one entry per interface, in kbps:

```
{
  "interfaces": [
    { "name": "GigabitEthernet4", "bandwidth_kbps": 50000 }
  ]
}
```

3. In **Test with a device**, pick the router and check column 3. What is highlighted comes from your parameters:

```
interface GigabitEthernet4
 bandwidth 50000
!
```

4. Save and close the cycle. The extension's state test checks that the interface stays up.

A new service with Argos from a sample config

You want an IP SLA probe toward the provider gateway on every branch router, and there is no service for it.

1. Open **Services → Custom services** and paste the configuration you already use on a device:

```
ip sla 10
 icmp-echo 198.51.100.1 source-interface GigabitEthernet2
 frequency 30
ip sla schedule 10 life forever start-time now
```

2. Argos proposes the template, the fields (for example, probe number, target, source interface and frequency) and a sample of values. Check that what varies is a field and what is fixed stays fixed.
3. Check the preview with two or three routers. If the gate rejects something, fix it before moving on.
4. Publish, declare the values in the right scope, for example `by_role`, and close the cycle.

IOS XR · A new BGP route reflector client

You are adding a new PE as a route reflector client and want to see the XML that changes before saving.

1. In **Services → Transport**, open the core BGP service (`xr_bgp`).
2. In **Test with a device**, pick the route reflector. Column 3 shows its XML, with the values highlighted.
3. In column 1, add the new PE's loopback to `clients`. Remember: a list is rewritten whole.

```
{
  "asn": 65000,
  "route_reflector": true,
  "clients": ["192.0.2.11", "192.0.2.12", "192.0.2.13"]
}
```

4. Click **Test these values**. Oktavia compares today's NETCONF plan with the plan for your change and shows the new neighbor:

```
+ <neighbor>
+   <neighbor-address>192.0.2.13</neighbor-address>
+   <remote-as>65000</remote-as>
+   <update-source>Loopback0</update-source>
+   <address-families>
+     <address-family>
+       <af-name>vpnv4-unicast</af-name>
+       <route-reflector-client/>
+     </address-family>
+   </address-families>
+ </neighbor>
```

5. If it is as expected, edit the declaration with **Edit values**, check **See what changes** and save.

6. The core changes as a single transaction with *commit confirmed*: if you don't confirm, the devices roll back on their own.

17. How templates and values work

Two pieces

Every service has a **template** per platform —Jinja for CLI, YANG XML for NETCONF— and its **values** in the intent (`service_vars`). The engineer only touches values; Oktavia maintains the template and it is not edited: it is extended.

Scopes and who wins

Values are declared per scope and merged from the most general to the most specific: `defaults`, `by_site`, `by_site_type`, `by_role`, `by_platform`, `by_model`, `by_tag`, `by_filter` and `by_device`. The last one wins; objects merge field by field and lists are replaced whole. The full table is in the Intent guide.

`by_filter` combines conditions, and it also reads the **key = value attributes** of the device and its site: `attr.tier=gold__role=pe` applies to PEs with that attribute and `site_attr.region=north` to devices in the sites of that region. When two filters apply, the one with more conditions wins.

Extensions

An extension declares `extends` with the base service, its fields, its template and its removal template. It generates its own block: it does not rewrite the base template or another service's lines (the gate prevents it). Its state or value test reaches compliance with it.

Removals

Removing something from the intent does not delete it from the device by itself. That is why every extension has its removal template (`no …` on CLI) and, on IOS XR, the removed row is marked `retired` and the plan removes it with `nc:operation="remove"`; compliance expects it gone.

From value to device

Saving does not touch the network. The change arrives through the cycle: the Digital Twin proposes the diff, a person approves, it is applied (on IOS XR with prior validation and confirmed commit) and the audit verifies it.

18. With Ansible and the API

The same can be automated from outside: a playbook can change a parameter or add a customer service with the preview, save the intent and return the change proposed in the Twin; approving and applying are still a person's call. Step by step in Automate Oktavia with its API, Ansible and Python.

19. Good practices

- **Test before saving.** "Test these values" saves nothing: use it freely.
- **Read "See what changes" device by device.** If a device you didn't expect appears, check the scope.
- **Prefer the catalog.** A catalog extension is already tested and ships with its network test.
- **Extend, don't copy.** The base template is not edited; your work lives in the extension and the values.
- **Declare at the most general scope possible.** Same as with intent: `by_device` only for what belongs to the device.
- **Give every extension its test.** A service with "no tests" won't warn you when it stops working.
- **No secrets in clear text.** Everything through the vault, with `<<vault:…>>`.
- **Verify with compliance.** A change ends when the audit confirms it.
- **Use attributes for groups that are neither site nor role** (customer tier, region, contract) and declare them with `by_filter` instead of repeating values per device.
- **Name extensions well.** The id says the service and what it adds (`xr_bgp_timers`, `ospf_bfd`); the label, in network language.
- **Bump the version when you change a published extension.** Oktavia keeps the version and its catalog origin; compliance and the Twin always use the published one.
- **On the core, validate first.** On IOS XR test the change (dry-run) and apply with confirmed commit: if verification fails, the device rolls back on its own.

For scopes and the change cycle, continue with the Intent guide.

About Oktanet

Oktanet is a Mexican network automation company focused on controlled, traceable, and scalable operations for enterprise environments in Latin America.

For a demonstration on your own environment, reach us at oktanet.io.

LinkedIn · **YouTube**

© 2026 Oktanet · Oktavia · Public document, with no confidential customer information.