ChatGPT

Segment Routing PCE in Cisco Crosswork Network Controller 7.1

Cisco Crosswork Network Controller (CNC) 7.1 integrates a **Segment Routing Path Computation Element (SR-PCE)** to provide centralized traffic engineering control for both MPLS and SRv6 networks. In this report, we detail how SR-PCE functions within CNC's architecture, the types of tunnel provisioning supported (RSVP-TE, SR-MPLS, and SRv6), and the northbound REST APIs used to create and manage tunnels. We also cover constraint-based path computation (affinities, disjointness, bandwidth, delay, protection), how to use Postman to test these APIs (including JWT authentication), and provide example Python script snippets for end-to-end automation (authenticate, create tunnels, and validate outcomes). Finally, we illustrate use-case topologies, diagrams, and sample outputs to contextualize these concepts. The focus is entirely on **external interfaces (northbound APIs)** and usage – internal CNC implementation details are minimized.

Architecture of SR-PCE in CNC 7.1 (Components & Interactions)

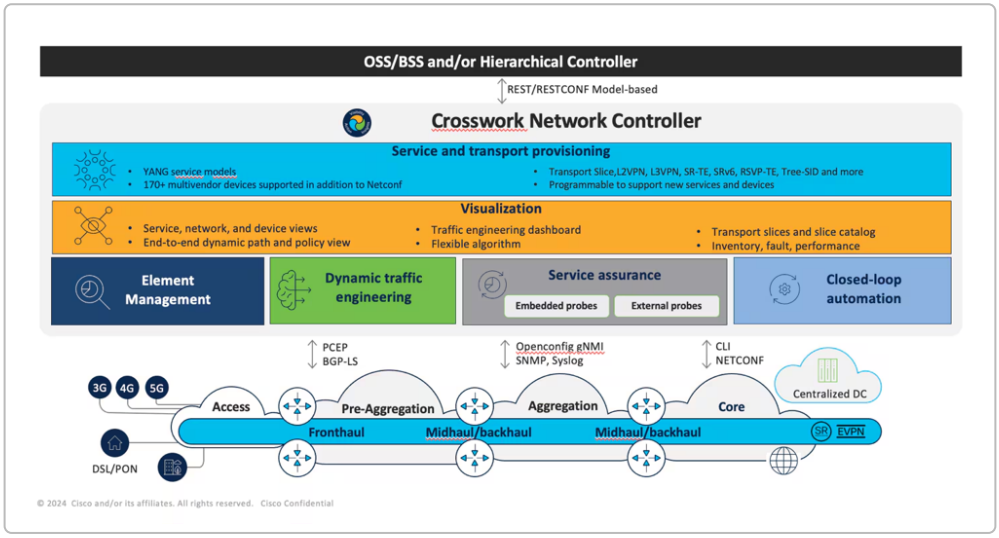

Figure 1: High-level CNC Architecture (Crosswork Network Controller with SR-PCE and related components). The SR-PCE provides traffic engineering for SR-MPLS, SRv6, and RSVP-TE, communicating with network devices via BGP-LS (topology discovery) and PCEP (path setup) [1] [2] .

At its core, Cisco CNC is a cluster-based platform that hosts various microservices and functions (device management, telemetry, provisioning, etc.). The **SR Path Computation Element (SR-PCE)** is a key component for traffic engineering in CNC 7.1. Rather than being an internal algorithm running on the

controller, the SR-PCE functionality is provided by an IOS-XR based PCE engine (which can run on an XR router or virtual XR instance) integrated with CNC [2] [3]. Key architectural points include:

- **IOS-XR Stateful PCE**: CNC relies on a stateful PCE (running IOS-XR software) that supports multi-domain operation and both Segment Routing and RSVP-TE. The SR-PCE can compute paths across IGP domains or ASes, as long as it has a complete topology view of the network [4] [5]. Multi-AS or multi-domain path computation is supported if each PCE instance has the full network topology (one PCE cannot compute on partial topology unknown to it) [4].

- **Topology Discovery**: The PCE learns the network topology and TE attributes through the IGP (OSPF/ISIS with Segment Routing extensions) or via BGP-LS. It collects all nodes, links, metrics, SR SIDs, etc., building a traffic engineering database [2]. CNC 7.1's SR-PCE uses BGP-LS and streaming telemetry to obtain a real-time view of the network state [1].

- **Path Computation and Installation**: The PCE computes end-to-end paths that meet specified constraints or SLA requirements (e.g. minimize latency or TE metric, avoid certain links, meet bandwidth needs). It is a **stateful PCE** – network devices (acting as PCCs, Path Computation Clients) report their existing tunnels to the PCE and can delegate control of tunnels to the PCE [6]. The PCE can **instantly program new tunnels** or update paths using PCEP (Path Computation Element Communication Protocol): it sends the computed list of segments (SID labels or SRv6 SIDs) to the head-end router via PCEP to instantiate an SR Policy, or it can initiate an RSVP-TE LSP setup by instructing the head-end. [1] [6] In SR-MPLS or SRv6 scenarios, the PCE effectively pushes an SR policy with an ordered SID list to the headend; in RSVP-TE, the PCE may trigger the headend to signal an RSVP path (for example by providing an explicit route object).

- **Integration with CNC**: In CNC's architecture, the SR-PCE is treated as a **provider** of path computation and tunnel management services. CNC is configured with one or more SR-PCE instances (for redundancy or multi-domain scenarios). The SR-PCE connects to CNC via gRPC and HTTP/PCEP interfaces [7]. The CNC controller uses:

- **gRPC** to interface with the PCE for topology discovery and synchronization of SR policy state (this is how CNC learns existing SR-MPLS/SRv6 policies and network links from the PCE) [8].
- **PCEP (via TCP)** for actual tunnel signaling – the PCE uses PCEP sessions to the network devices (head-end routers) to install or remove TE tunnels (for both SR and RSVP). CNC provides credential profiles for these connections (e.g. basic HTTP text authentication for PCEP if required) [8].

In essence, CNC's northbound APIs (described below) act as an **intent-based interface**: when an external system or user requests a tunnel or policy creation via the API, CNC relays this request to the SR-PCE component, which computes the path and programs the network accordingly. CNC then reports back the result (success/failure) via the API response. CNC also continuously monitors the tunnels' status via telemetry (e.g., using streaming telemetry, SNMP, BGP-LS updates, SR Performance Monitoring, etc.) to provide visualization and closed-loop optimization.

- **Supported Tunneling Technologies**: The integrated SR-PCE supports provisioning of **Segment Routing Traffic Engineering (SR-TE)** policies in both MPLS and SRv6 data planes, as well as classic **RSVP-TE LSPs** [1] [5]. This means operators can manage RSVP-TE tunnels (e.g. for non-SR networks

or interop scenarios) and Segment-Routed paths (SR-MPLS or SRv6) using a unified controller. CNC 7.1's traffic engineering module visualizes all these tunnel types on the topology map [9].

In summary, **SR-PCE in CNC 7.1** serves as the brain for path computation and TE tunnel lifecycle. It bridges the CNC northbound intent (APIs or GUI requests to create a tunnel with constraints) to the actual network programming via PCEP, leveraging a global view of network state to compute optimal paths that satisfy the requested constraints [1]. It is a stateful, multi-protocol PCE that can handle RSVP-TE, SR-MPLS, and SRv6 concurrently.

Supported Tunnel Provisioning Types in CNC 7.1

CNC 7.1 (with SR-PCE) supports three main types of traffic-engineered tunnels/policies:

- **RSVP-TE (MPLS Traffic Engineering using RSVP)** – Traditional MPLS-TE tunnels with RSVP signaling. These are label-switched paths where resources can be reserved. CNC can orchestrate RSVP-TE tunnels via the PCE as a stateful controller (the PCE computes the route and can instruct the head-end router to signal the RSVP path). RSVP-TE tunnels can coexist with SR policies and are visible/manageable in the CNC UI and APIs [10] [11].

- **SR-MPLS Policy (Segment Routing MPLS)** – Segment Routing TE policy using MPLS label SIDs. An SR-MPLS policy is identified by a tuple (Head-end, Color, Endpoint), with an associated Binding SID (an MPLS label that acts as a handle for the policy in the data-plane). CNC (via the PCE) can create two flavors of SR-MPLS policies:

- *Dynamic path SR policy*: the PCE computes the path dynamically based on an objective (e.g. minimize IGP metric or delay) and constraints (affinities, disjointness, etc.), and it may reoptimize the path as network conditions change. No explicit hop list is provided by the user – only the intent (constraints/objective) is given, and the PCE picks the best route.

- *Explicit path SR policy*: the user specifies an explicit list of segments (SIDs) or hops for the path. This can be a list of node-SIDs and/or adjacency-SIDs that define the exact route the traffic should take. The PCE will program the policy with exactly this SID list (assuming it's viable). Explicit paths might be used for static routing requirements or to pin a specific path.

- **SRv6 Policy (Segment Routing over IPv6)** – Segment Routing policy using SRv6 SIDs (128-bit IPv6 segment identifiers). Similar to SR-MPLS, an SRv6 policy has a head-end, color, endpoint, and possibly a binding SID (in SRv6 the binding SID is an SRv6 SID that maps to the policy, typically automatically allocated from the headend's SRv6 locator). CNC supports SRv6 TE policies in both dynamic and explicit modes as well:

- *Dynamic SRv6 policy*: PCE computes an SRv6 path (a sequence of SRv6 SIDs representing node or adjacency segments) to satisfy the intent. The headend, after receiving the SID list from the PCE, instantiates an SRv6 policy (often as an ordered list of SRv6 SIDs in a routing header).
- *Explicit SRv6 policy*: A specific list of SRv6 SIDs (e.g. node SIDs or adjacency SIDs in IPv6 address form) is provided to define the path. The PCE validates and installs this exact SID list as the policy's path.

It's worth noting that the **northbound API model for SR policies** (covered in the next section) generally covers both SR-MPLS and SRv6. The type of policy (MPLS vs SRv6 data-plane) is usually inferred from the

context – typically by the **head-end router's capabilities and the addresses/SIDs provided**. For example, if the head-end device supports SRv6 and an SRv6 locator, CNC can create an SRv6 policy on it. In the CNC GUI, there is an "Enable SRv6" toggle when creating a policy, which indicates the policy will use SRv6 SIDs [12]. From an API perspective, using IPv6 addresses for head-end/endpoint and specifying SRv6 SIDs in the hops would result in an SRv6 policy.

All three tunnel types are managed under CNC's Traffic Engineering application. Operators can view them on the topology map (with SR-MPLS and SRv6 policies shown in their respective tabs, and RSVP-TE LSPs in an RSVP tab) [13] [14]. The controller keeps track of which tunnels are **PCE-initiated vs PCC-initiated**. For instance, an SR policy created via CNC is PCE-initiated; CNC also learns about any SR policies or RSVP tunnels that were locally configured on devices (PCC-initiated) if those devices report them to the PCE. Both are visible, but only PCE-initiated ones can be modified via the controller.

In summary, CNC 7.1's SR-PCE can provision **RSVP-TE LSPs**, **SR-MPLS policies**, and **SRv6 policies**. Next, we will explore the northbound RESTful APIs through which these tunnels are created and managed.

Northbound APIs for Tunnel Provisioning (RSVP-TE, SR-MPLS, SRv6)

CNC 7.1 exposes **RESTCONF** APIs (over HTTPS) for external systems to provision and manage TE tunnels. These northbound APIs are model-driven (YANG data models) and require JSON payloads. All API calls are authenticated via a JWT Bearer token (see the authentication section below). The base URL for CNC APIs is typically:

```
https://{CNC_host}:{port}/crosswork/  (e.g., port 30603 by default)
```

Specific API endpoints exist for different operations (creating, deleting tunnels, etc.). Under the hood, these correspond to RESTCONF operations on CNC's "Optimization Engine" service. Below, we detail the **full API paths, payload schema, required fields, and example responses** for each requested operation:

API for Creating an RSVP-TE Tunnel (MPLS RSVP-TE)

Endpoint: `POST /crosswork/nbi/optimization/v3/restconf/operations/cisco-crosswork-optimization-engine-rsvp-te-tunnel-operations:rsvp-te-tunnel-create`

This API creates a new RSVP-TE tunnel (LSP) in the network via the SR-PCE. The HTTP request body must be JSON and conform to the YANG model for RSVP-TE tunnel creation. The key fields and structure are:

- **head-end (string, required):** IP address of the tunnel's source (head-end router) [15].
- **end-point (string, required):** IP address of the tunnel's destination (tail-end router) [15].
- **path-name (string, required):** A name or identifier for the tunnel's path [16]. This is an arbitrary user-defined name for reference (it may correspond to the tunnel name on devices).
- **description (string, optional):** Description of the tunnel (free text).
- **binding-label (integer, optional):** A binding label for the RSVP-TE tunnel [17]. (This is used if the RSVP tunnel is to be used as a segment in SR or for stitching purposes. It can usually be omitted unless a specific label is needed.)

- **setup-priority / hold-priority (integers, optional):** RSVP-TE setup and hold priority (0-7, where 0 is highest priority) [18] . These default if not set (typically default 7 if unspecified).
- **signaled-bandwidth (integer, optional):** The bandwidth to reserve (in bytes per second) for the tunnel [19] . Default is 0 if not reserving bandwidth.
- **fast-re-route (string, optional):** Enable or disable fast reroute (link/node protection) for this LSP [20] . Allowed values: `"enable"` or `"disable"` (default typically disable if not provided).
- **rsvp-te-tunnel-path (object, required):** This object defines the path specification. It can represent either a *dynamic* path or an *explicit* path:
- For a **dynamic path** (constraint-based routing by PCE), you do **not** include an explicit hops list. Instead, you may specify:
    - **optimization-objective (string, optional):** The metric to minimize. Choices include `"igp-metric"`, `"te-metric"`, or `"delay"` [21] . (E.g., minimize IGP cost, TE metric, or cumulative delay).
    - **affinities (object, optional):** Link affinity constraints (coloring constraints) [22] . You can specify bitmasks for:
    - `include-all` – links must have all these affinities,
    - `include-any` – links must have at least one of these affinities,
    - `exclude-any` – links must not have any of these affinities. These are provided as unsigned integer bitmasks representing the combination of attribute flags [22] .
    - **disjointness (object, optional):** Path disjointness constraint [23] . This is used if the new tunnel should be disjoint (diverse) from another tunnel or group of tunnels. You provide:
    - `association-group` (integer ID) and `association-sub-group` (integer) to identify a group of tunnels,
    - `disjointness-type` (string) to specify the type of disjointness: can be `"node"`, `"circuit"` (link), `"srlg"`, or combined `"srlg-node"` [24] . For example, if you want this LSP to be link-disjoint from another, you'd use the same association group on both and set type to circuit.
    - *(If needed)* **bandwidth**: In the RSVP model, bandwidth is directly the `signaled-bandwidth` field above, not part of the path object – whereas for SR (discussed later) there is a separate notion of bandwidth-aware path. RSVP dynamic path inherently considers reservable bandwidth if `signaled-bandwidth` is set.
    - The absence of an explicit hop list triggers the PCE to run CSPF (Constrained Shortest Path First) based on the specified constraints to find a suitable path.
- For an **explicit path**, you include a **hops** list inside `rsvp-te-tunnel-path` and omit the dynamic parameters:
    - **hops (array):** An ordered list of hop objects representing the path [25] .
    - Each hop has a `hop-address` (either a node or adjacency address) and a `hop-type` ("strict" or "loose") [26] .
    - In the model, this is structured such that each hop entry has a `hop` sub-object, which can contain either:
        - `node-address` (IPv4 address of a node to go via) **or**
        - `adjacency-address` (IPv4 address of the next-hop interface) [27] .
    - You also specify a `step` (sequence number for the hop order) and a `hop-type` indicating if it's a strict or loose hop [28] . "Strict" means the next hop must be directly connected, "loose" means it can be reached via any route as long as ultimately the path goes through that node.
    - If `hops` are provided, the PCE will install an explicit route object (ERO) for the RSVP-TE LSP matching this sequence.

- In this case, *do not* include optimization-objective or affinities (those are ignored if an explicit path is given).

Example Request Payload (RSVP-TE, dynamic path):

```
{
  "input": {
    "rsvp-te-tunnels": [
      {
        "head-end": "192.0.2.1",
        "end-point": "192.0.2.5",
        "path-name": "LSP1",
        "signaled-bandwidth": 50000000,
        "setup-priority": 7,
        "hold-priority": 7,
        "rsvp-te-tunnel-path": {
          "optimization-objective": "te-metric",
          "affinities": {
            "exclude-any": 2
          },
          "disjointness": {
            "association-group": 10,
            "association-sub-group": 0,
            "disjointness-type": "node"
          }
        }
      }
    ]
  }
}
```

This example would ask the PCE to create a tunnel from `192.0.2.1` to `192.0.2.5` named "LSP1", reserving ~50 Mbps, with default priorities. It instructs the PCE to compute a path that minimizes TE metric, excludes links with affinity bit 1 (since exclude-any=2 which is binary `10`), and is node-disjoint from other tunnels in group 10. No explicit hops are provided, so the PCE will calculate the best path meeting those constraints.

Example Request Payload (RSVP-TE, explicit path):

```
{
  "input": {
    "rsvp-te-tunnels": [
      {
        "head-end": "192.0.2.1",
        "end-point": "192.0.2.5",
        "path-name": "LSP2",
```

```
        "fast-re-route": "enable",
        "rsvp-te-tunnel-path": {
          "hops": [
            {
              "hop": {
                "node-address": "198.51.100.10"
              },
              "step": 1,
              "hop-type": "loose"
            },
            {
              "hop": {
                "node-address": "198.51.100.20"
              },
              "step": 2,
              "hop-type": "loose"
            }
          ]
        }
      }
    ]
  }
}
```

In this example, we create “LSP2” from 192.0.2.1 to 192.0.2.5, explicitly routing via node 198.51.100.10 then 198.51.100.20 (two intermediate hops). Both hops are marked loose (meaning any route to those nodes is acceptable). Fast Reroute (FRR) is enabled for link protection on this LSP. The PCE will install this LSP with the given explicit route.

Response: On success, the API returns HTTP 200 with a JSON body indicating the outcome. The response model contains an `output` with a `results` list. Each result will echo the head-end, end-point, and path-name, and give a **state** and message:

- **state:** `"success"` if the tunnel was created successfully, `"failure"` if it failed, or `"degraded"` if partially successful (e.g., created but not with all constraints) [29].
- **message:** human-readable message, especially useful on failure (e.g., reason for failure) [30].

For example, a successful creation might return:

```
{
  "output": {
    "results": [
      {
        "head-end": "192.0.2.1",
        "end-point": "192.0.2.5",
        "path-name": "LSP1",
```

```
        "state": "success",
        "message": "Tunnel created"
      }
    ]
  }
}
```

If an error occurred (say, no path found satisfying constraints), `state` would be "failure" and the message might describe the constraint violation or network issue.

API for Creating an SR-MPLS Policy (Segment Routing MPLS TE)

Endpoint: `POST /crosswork/nbi/optimization/v3/restconf/operations/cisco-crosswork-optimization-engine-sr-policy-operations:sr-policy-create`

This API creates a new Segment Routing policy (SR-TE policy) with MPLS data plane (label stack). The SR policy is identified by the head-end, color, and endpoint combination. The payload structure has many similarities to the RSVP-TE case, but with SR-specific fields:

- **head-end (string, required):** IP address of the policy's source (head-end router) [31].
- **end-point (string, required):** IP address of the policy's destination (explicit endpoint of the traffic engineering policy) [31].
- **color (integer, required):** The SR policy color (also sometimes called "policy ID") [32]. This is an arbitrary number used to identify the policy on the head-end (and match against steering rules). Color together with endpoint must be unique per head-end. (For example, color 100 might correspond to "gold traffic" policy).
- **path-name (string, optional):** Name of the path (or candidate path). In the context of SR, a single SR Policy can have multiple candidate paths (each with its own name). If you are only creating one path for this policy, you can still give it a name (e.g. "primary"). In some workflows this might be optional.
- **binding-sid (integer, optional):** The binding SID (BSID) for the SR policy [33]. For SR-MPLS, this is the MPLS label value that will be used as the policy's BSID. If not provided, the network (head-end) might allocate one or a default could be used. Typically this is optional; if you want a specific label as BSID you can provide it.
- **profile-id (integer, optional):** A profile ID for the SR policy [34]. Profile might refer to a predefined set of policy parameters or template in the controller. In many cases this can be omitted unless profiles are defined.

- **description (string, optional):** Text description of the policy [34].

- **sr-policy-path (object, required):** Defines the path details. This has sub-choices similar to RSVP:

- **Dynamic path (automated path computation)** – Indicated when using the `dynamic-path` fields:
  - **path-optimization-objective (string, optional):** Objective to minimize, choices: `"igp-metric"`, `"te-metric"`, `"delay"`, or `"hop-count"` [35]. (Hop-count means minimize number of hops).

- **affinities (object, optional):** Affinity constraints for links (include/exclude affinities), same structure as in RSVP (include-all, include-any, exclude-any as 64-bit integers) [36] .
- **disjointness (object, optional):** Disjointness constraints, similarly with `association-group`, `association-sub-group`, and `disjointness-type` ("node", "circuit", "srlg", "srlg-node") [37] . This allows making the SR policy path disjoint from other policy or LSP groups (requires those tunnels to share an association group ID).
- **protected (boolean, optional):** Whether to use only protected links in computation [38] . If `protected=true` (default), the PCE will prefer paths that use protected adjacency SIDs (i.e., links with fast reroute backup) [38] . Setting this to false allows unprotected links.
- **sid-algorithm (integer, optional):** Flex-Algo identifier to consider if computing the path using a specific SR Flex-Algo (if applicable) [39] . If not using Flex-Algo, this can be omitted or 0.
- **bandwidth (integer, optional):** Desired bandwidth for the SR policy (in some unit, likely bytes/sec or bps, typically 32-bit) [40] . If specified, it triggers a *bandwidth-aware* path computation. This is actually a separate sub-mode:
- **bw-path-optimization-objective (string, optional):** Objective for the bandwidth-aware path, choices similar to above (igp-metric, te-metric, delay, hop-count) [41] . Essentially, if you specify a bandwidth, the PCE will find a path that can accommodate that bandwidth (looking at live traffic/utilization or reservable bandwidth) and optimize for the given metric. This might internally use a different CSPF that considers available bandwidth on links (making use of live utilization data via telemetry, if supported).
- *Note:* The model indicates that *either* a normal dynamic path or a bandwidth-aware path can be specified. If you include the `bandwidth` field, you should use the `bw-path-optimization-objective` instead of the normal `path-optimization-objective`. These are marked by `x-choice` in the YAML model [42] [41] , meaning they are mutually exclusive options.
- With a dynamic path, no explicit SID list is given – the PCE will calculate it.

- **Explicit path** – Indicated by providing the `hops` list under `sr-policy-path`:
  - **hops (array):** List of SID hops in order [43] [44] . Each hop is defined similarly to RSVP but with support for segment types:
  - Each hop entry has a `hop` object that can specify one of several SID types:
    - `node-ipv4-address` / `node-ipv6-address`: the address of a node SID (for SR-MPLS, typically the prefix that has a Node SID; for SRv6, an IPv6 locator or loopback of the node) [45] .
    - `node-ipv4-sid` / `node-ipv6-sid`: if you know the SID value of a node SID (e.g., an index or absolute SID value), you could specify it. Typically one would use the address rather than numeric SID for node, so these might be less used.
    - `adjacency-ipv4-address` / `adjacency-ipv6-address`: the actual address of an adjacency (next-hop interface address) if you want to include a specific link [43] .
    - `adjacency-ipv4-sid` / `adjacency-ipv6-sid`: numeric SID value for an adjacency (for SR-MPLS, the adjacency SID label; for SRv6, possibly the function portion of an adjacency SID). Usually specifying by address is more straightforward.
  - Essentially, you provide either an IP address or a SID value for each hop, and the system will interpret it. For SR-MPLS explicit paths, you might list node IPv4 addresses (loopbacks) for each hop – the PCE will translate those to the corresponding node-SIDs (labels). For SRv6 explicit, you would likely list the node IPv6 addresses (SRv6 locators of each hop) or adjacency addresses.

- The hops array also may include a `step` (to sequence them, though order in array is usually enough) and is marked as an explicit path choice [46].
- When `hops` are provided, you should not include dynamic fields like optimization objective or affinities; the presence of `hops` signals this is an explicit user-defined SID list.

- The SR model also allows multiple candidate paths under a single policy (each with its own path-name). However, the create operation typically provisions one policy with one candidate at a time. If multiple candidate paths are needed (e.g., a primary and a backup explicit path), one would issue separate API calls or use an extended payload to add additional candidate-path entries.

Example Request Payload (SR-MPLS, dynamic path):

```
{
  "input": {
    "sr-policies": [
      {
        "head-end": "192.0.2.1",
        "end-point": "192.0.2.8",
        "color": 100,
        "path-name": "LowDelay",
        "binding-sid": 24000,
        "description": "Low-latency path PE1-PE2",
        "sr-policy-path": {
          "path-optimization-objective": "delay",
          "protected": true,
          "affinities": {
            "include-any": 1
          },
          "disjointness": {
            "association-group": 20,
            "association-sub-group": 0,
            "disjointness-type": "srlg"
          }
        }
      }
    ]
  }
}
```

This asks for a new SR policy from 192.0.2.1 to 192.0.2.8 with color 100. We name the candidate path "LowDelay" and even specify a desired binding SID of 24000 (an MPLS label). The PCE is told to compute a path optimizing for minimal delay, only using *protected links* (`"protected": true`), and including at least one link affinity bit 0x1 (`include-any: 1` which could, for example, mean "only use gold links"). We also request SRLG-disjointness from association group 20 (perhaps ensuring this path is disjoint from another policy in group 20). No explicit SID list is given, so PCE will choose the SID sequence.

Example Request Payload (SR-MPLS, explicit path):

```
{
  "input": {
    "sr-policies": [
      {
        "head-end": "192.0.2.1",
        "end-point": "192.0.2.8",
        "color": 200,
        "path-name": "ExplicitPath1",
        "binding-sid": 24001,
        "sr-policy-path": {
          "hops": [
            {
              "hop": { "node-ipv4-address": "198.51.100.10" },
              "step": 1
            },
            {
              "hop": { "node-ipv4-address": "198.51.100.20" },
              "step": 2
            },
            {
              "hop": { "node-ipv4-address": "198.51.100.30" },
              "step": 3
            }
          ]
        }
      }
    ]
  }
}
```

Here we create another policy from 192.0.2.1 to 192.0.2.8 (head-end and endpoint the same as before, but using a different color 200 to distinguish it). We specify `ExplicitPath1` with an explicit route through three intermediate nodes (with loopback IPs 198.51.100.10 -> .20 -> .30). We gave a BSID 24001. The PCE will install a segment list corresponding to these three hops (each will translate to the node-SID label of those nodes on the path from PE1 to PE2). This is a strictly explicit route; we did not request any particular optimization or constraint beyond the hops given.

Response: The SR policy create API returns a similar structure with `output.results`. Each result includes the head-end, end-point, color, and the outcome:

```
{
  "output": {
    "results": [
      {
        "head-end": "192.0.2.1",
```

```
        "end-point": "192.0.2.8",
        "color": 100,
        "state": "success",
        "message": "Policy created"
      }
    ]
  }
}
```

If for example the path couldn't be instantiated, `state` might be "failure" and message could say "no path satisfying constraints (affinity, disjointness) found", etc. A degraded status might occur if, say, the policy was created but not all constraints were honored (CNC would then flag that as degraded).

API for Creating an SRv6 Policy (Segment Routing IPv6 TE)

The API for SRv6 policy creation is **the same endpoint as SR-MPLS** (`sr-policy-create`), as it uses the same model. There is not a separate endpoint; instead, the SRv6 vs SR-MPLS distinction is contextual. To create an SRv6 policy, ensure that the head-end device supports SRv6 and indicate that in the request:

- Use the **head-end's IPv6 address** (often the router ID or loopback if defined as an IPv6) in the head-end field, and similarly an IPv6 address for the endpoint if applicable.
- Provide an SRv6 Binding SID if needed. Note: In many SRv6 deployments, the head-end allocates the BSID from its SRv6 locator. The API model's `binding-sid` is an integer, which suited MPLS labels. For SRv6, the BSID is an IPv6 address, which the model does not explicitly ask for. This implies that typically the PCE/head-end will choose the SRv6 BSID automatically (likely from a locator and the color as input). In the CNC UI, enabling "SRv6" on a policy likely triggers the headend to use an SRv6 BSID.
- In the hops, use IPv6 addresses if specifying explicit segments (e.g., use `node-ipv6-address` or `adjacency-ipv6-address`). If dynamic, the PCE will compute the SRv6 SID list.

Essentially, **the payload for SRv6 looks identical to SR-MPLS** in structure. Just the actual addresses/SIDs differ. For instance, an explicit SRv6 policy path might list node IPv6 addresses for each hop. The PCE then knows it's dealing with SRv6 SIDs (since those nodes advertise SRv6 SID information via BGP-LS). The `sr-policy-path` object has fields for IPv6 adjacency and node SIDs which we saw in the model [43] [44]. Also, the PCE being "SRv6-aware" means it can handle SRv6 segments [5].

Example Request Payload (SRv6, dynamic path):

```
{
  "input": {
    "sr-policies": [
      {
        "head-end": "2001:db8:0:1::1",
        "end-point": "2001:db8:0:5::1",
        "color": 101,
        "description": "SRv6 latency-optimized path",
```

```
        "sr-policy-path": {
          "path-optimization-objective": "delay",
          "protected": true
        }
      }
    ]
  }
}
```

Here, the head-end and end-point are IPv6 addresses, implying an SRv6 policy from those routers. We left out binding-sid (so it will be auto-chosen) and just specified we want a low delay path with protected links. The response would indicate success if an SRv6 policy was created. (The head-end would allocate an SRv6 BSID, typically an address in its locator with some policy indicator.)

Example Request Payload (SRv6, explicit path):

```
{
  "input": {
    "sr-policies": [
      {
        "head-end": "2001:db8:0:1::1",
        "end-point": "2001:db8:0:5::1",
        "color": 201,
        "path-name": "ExplicitSRv6",
        "sr-policy-path": {
          "hops": [
            { "hop": { "node-ipv6-address": "2001:db8:0:10::1" }, "step": 1 },
            { "hop": { "node-ipv6-address": "2001:db8:0:20::1" }, "step": 2 }
          ]
        }
      }
    ]
  }
}
```

This requests an SRv6 policy from `2001:db8:0:1::1` to `2001:db8:0:5::1` (perhaps PE1 to PE2) with a specified explicit path going via nodes with IPv6 loopbacks `2001:db8:0:10::1` then `2001:db8:0:20::1`. The PCE will respond by programming an SRv6 policy on PE1, where the SID list contains the SRv6 Node-SIDs of those intermediate nodes in that order. The color 201 differentiates it from other policies. If the headend has SRv6 enabled, it will use an SRv6 BSID (not directly specified here, likely automatically assigned).

Response: Same structure – listing head-end, end-point, color, and state/message in results. For example:

```
{
  "output": {
    "results": [
      {
        "head-end": "2001:db8:0:1::1",
        "end-point": "2001:db8:0:5::1",
        "color": 201,
        "state": "success",
        "message": "Policy created"
      }
    ]
  }
}
```


The above Northbound API operations allow full control of tunnel provisioning. Deletion or modification operations also exist (e.g., there are corresponding `...:rsvp-te-tunnel-delete` and `...:sr-policy-delete` endpoints, and even an **"update" might be done by re-posting create with same identifiers or via a different RPC** – but those are beyond scope here). There is also a **"dry-run"** capability (for SR policies, an API to preview the route without actually instantiating the policy) [47], which can be useful for checking what path *would* be taken.

Next, we discuss how to specify various *constraints* and advanced parameters in these API calls in more detail.

Constraint-Based Path Computation in SR-PCE (Affinity, Disjointness, Bandwidth, Delay, Protection)

One of the strengths of using a PCE-driven controller is the ability to request paths that meet specific constraints or optimization criteria. As seen in the API schemas, CNC's NB APIs allow the client to specify a variety of constraints for dynamic path computation:

- **Link Affinity Constraints (Include/Exclude Colors):** Network links (TE links) often have administrative attributes (affinity bits or "colors"). For example, some links might be tagged as `blue` (maybe high-bandwidth optical links) vs `green` (lower capacity links), etc. In the API, the `affinities` object lets you filter links based on these bits.
- `include-all`: the path must traverse only links that have **all** of these bits set.
- `include-any`: the path must have at least one of these bits on each link.
- `exclude-any`: the path must **not** use any link that has any of these bits. These are provided as numeric bitmasks in the API [22]. For example, if color bit 0 represents "satellite link", you might set exclude-any = 1 (binary ...0001) to avoid all satellite links. Affinity constraints are supported for both RSVP-TE and SR policies [22] [36]. In Crosswork UI, these correspond to TE link affinity names; in the API you use the bit values (which require knowing the bit assignments). Operators can configure TE link affinities in CNC or on devices and CNC will be aware of them [48] [49].

- **Disjointness Constraints (Path Diversity):** Sometimes you need two (or more) tunnels that are diversely routed (for redundancy). The PCE supports computing disjoint paths by using the **association group** mechanism. In the API, `disjointness` allows you to specify an association group ID and sub-group, and a disjointness type [23] [37]. For example, if you have a primary LSP and you want a secondary LSP that is link-disjoint from it, you could:

- When creating the primary LSP, assign it to group X (association-group = X, sub-group maybe 0).

- When creating the secondary LSP, also assign association-group = X but disjointness-type = "circuit" (link) or "node" as needed. The PCE will ensure the new path shares no links (or nodes) with any other tunnels in that group [24]. Disjointness can be specified for SR policies similarly (the group can include RSVP and SR tunnels as well, since PCE sees them all). There's also an SRLG-based disjointness option (srlg or srlg-node) where the PCE avoids Shared Risk Link Groups overlaps. This is useful for fiber-diverse routing (ensuring paths don't share a common fiber or conduit if known SRLG info is available) [50]. In practice, you generate a unique group ID and use it for tunnels that should be mutually disjoint. CNC's PCE then solves a disjoint path computation problem (it might compute both simultaneously if both are new, or one relative to the existing one). This is a powerful feature for reliability use-cases.

- **Bandwidth Constraints:** The PCE can compute paths that satisfy a bandwidth requirement if it has visibility into link capacities and current usage. In CNC 7.1, this is supported especially for SR policies via the `bandwidth` field in the SR policy API [51]. When you specify a bandwidth (e.g., in bps) and use a bandwidth optimization objective, the PCE will attempt to find a path that has at least that much unreserved or available bandwidth (using traffic stats or RSVP reservations). This effectively turns the PCE into a **bandwidth broker**, picking a path that can accommodate the traffic. CNC's release notes highlight that the controller can act as a bandwidth broker and perform path computations considering requested or measured bandwidth [52] [53]. For example, if you request 1 Gbps and the primary links are congested, PCE might choose a longer path that has enough free capacity. For RSVP-TE tunnels, bandwidth constraints tie into RSVP's admission control (the PCE will only choose a path where RSVP can reserve the bandwidth). For SR, since no RSVP reservations occur, CNC likely uses live utilization (from streaming telemetry) to judge if a link has room for that bandwidth (this is sometimes referred to as SR Dynamic Traffic Engineering where the controller avoids congested links). The API's separation of normal vs bandwidth-aware path (with `bw-path-optimization-objective`) ensures that if bandwidth is specified, the PCE uses the right metric (possibly a combined metric weighting utilization).

- **Delay Constraints:** Optimizing for latency is a common requirement for SR-TE. The PCE can minimize cumulative delay if it has per-link latency measurements. CNC 7.1 integrates **SR-PM (Segment Routing Performance Monitoring)** which measures delay on links using telemetry (TWAMP or RFC 6374 PM) [54] [55]. If delay metric is available, setting `optimization-objective: "delay"` will cause the PCE to choose the path with least end-to-end latency [42]. This is how low-latency paths (like for 5G fronthaul) can be computed. If no delay data is present, PCE would fall back to IGP metrics or might not differ from a normal shortest path. There isn't a hard "max delay" constraint in the API, but you can achieve similar effect by metric optimization or by using SR-PM to monitor and then disallow certain links via other means (like affinities marking high-latency links). Also note, CNC's Service Health and SR-PM features allow continuously monitoring delay of TE tunnels after they're set up [56].

- **Protection Constraints:** The term “protection” in context of path computation can mean:

- Use only links that have fast reroute (FRR) backup available (so that if a link fails, local repair happens within <50ms). In the SR policy API, this is the `protected` flag [38]. By default it is true, meaning the PCE will try to use only protected links for the computed path. If set to false, it may pick an unprotected link if it’s otherwise optimal. This is a proactive way to ensure reliability of the chosen path.
- Enabling **fast reroute** on the tunnel itself. In RSVP, we saw `fast-re-route: enable` which turns on RSVP local repair (one-hop bypass LSPs) [20]. In SR, link protection is inherent if using TI-LFA on the network, but the controller can decide whether to consider it.

- End-to-end 1+1 protection or secondary paths – not directly specified in the single create API, but one could create two SR policies (primary and standby) and steer traffic into primary with automatic fallback. CNC does support computing **Multiple Candidate Paths (MCP)** for an SR policy [57], effectively allowing a secondary path in the same policy (only one active at a time). This might be done through either multiple API calls or a combined payload; in UI it’s done via adding a candidate path. Disjointness can be used to ensure primary and backup are disjoint. The “active” path is indicated by PCE based on preference or metric and can automatically switch if one fails (if using PCE delegation).

- **Hop Limits or Explicit Partial Paths:** If needed, one can also impose partial explicit routes and let PCE fill the rest by using loose hops. For example, you could specify two loose hops (must go via NodeA and NodeB, in that order) but leave the segments between them to PCE. This is supported by combining explicit hops (with hop-type=loose) and dynamic computation in between. The PCE will treat those as constraints to route through those waypoints.

To summarize, CNC’s SR-PCE can handle advanced TE constraints, enabling **intent-based requests** like "Find me a node-and-SRLG-disjoint backup path that can carry 10Gbps and avoid satellite links, with under 50ms latency". The northbound API expresses these in the fields we discussed (affinity masks, disjointness groups, bandwidth and metric objective, etc.). If the constraints are unsatisfiable, the PCE will return a failure (and CNC would not instantiate a policy); if they are satisfiable, it will pick the best path that meets all constraints (or in some cases degrade one – e.g., perhaps meet everything except one affinity if marked optional – but typically it either meets or fails).

CNC’s path computation also works in conjunction with its **closed-loop optimization**: for example, the Link Congestion Mitigation (LCM) function in CNC can automatically create SR policies when a link is congested, moving traffic off that link [58]. In doing so, it uses the same PCE mechanism to find alternate paths. Constraints ensure that the new path indeed alleviates congestion (e.g., avoids that link).

Using the APIs with Postman (Authentication and Examples)

Before we can call any of the CNC northbound APIs, we must handle **authentication**. CNC 7.1 uses a JWT (JSON Web Token) based authentication scheme [59]. The process is slightly involved, as it uses Cisco’s Single Sign-On mechanism. The high-level steps are:

1. **Obtain a Ticket-Granting Ticket (TGT)** by posting your CNC user credentials.
2. **Exchange the TGT for a JWT** for API usage.

3. **Use the JWT in Authorization header** for subsequent API calls.

In Postman, you would typically do step 1 and 2 as separate requests (or use a scripting feature to automate). Here are the details:

- **Step 1: Get TGT** – Send an HTTP POST to the TGT endpoint with form-encoded body containing username and password. For example:
- **URL:** `https://{cnc_host}:30603/crosswork/sso/v1/tickets`
- **Headers:** `Content-Type: application/x-www-form-urlencoded`, `Accept: text/plain` [60]
- **Body:** `username=<your_username>&password=<your_password>` (as form data).

Example (Postman or curl):

```
POST https://10.0.0.1:30603/crosswork/sso/v1/tickets
Content-Type: application/x-www-form-urlencoded
Accept: text/plain

username=admin&password=YourPassword123
```

If the credentials are correct, the response will be a plain text ticket string (TGT). For example: `TGT-34-dDGV8oqlqLdDgRFlldWGPOui-XoRa8bcJ80XdQOAFqeAUYHvmMTFwm6kez35IcRmWQA-...` [61]. This is a one-time ticket (think of it like a session identifier).

- **Step 2: Get JWT** – Now take the TGT from step 1 and POST it to the JWT endpoint along with a service identifier:
- **URL:** `https://{cnc_host}:30603/crosswork/sso/v2/tickets/jwt`
- **Headers:** `Content-Type: application/x-www-form-urlencoded`
- **Body:** `tgt=<TGT_received>&service=https://{cnc_host}:30603/app-dashboard` [62].

Example:

```
POST https://10.0.0.1:30603/crosswork/sso/v2/tickets/jwt
Content-Type: application/x-www-form-urlencoded

tgt=TGT-34-dDGV8oqlqLdD... (the TGT string)
&service=https://10.0.0.1:30603/app-dashboard
```

The `service` parameter might seem odd, but it's required by the SSO to issue a token for a particular service (here we use the CNC dashboard URL as the target service). The response to this call, if successful, is

your **JWT token** (a long Base64-encoded string typically beginning with `eyJhbGciOiJI...`) [63]. For example:

```
eyJhbGciOiJIUzUxMiIsInR5cCI6IkpXVCIsImtpZCI6IjE0ZWQ0NGQwLTZhYTQtNDg4Yy05ZDI3LWFhNWVkMjE3OTcwMyJ9.
```

Copy this entire JWT string.

- **Step 3: Use JWT in API calls** – Now that you have the JWT, for any subsequent API request to CNC (like the tunnel creation APIs we discussed), set the `Authorization` **header to** `Bearer <JWT>` [64]. For example:

```
Authorization: Bearer
eyJhbGciOiJIUzUxMiIsInR5cCI6IkpXVCIsImtpZCI6IjE0ZWQ0NGQwLTa...<etc>
```

Also, ensure you include `Content-Type: application/yang-data+json` (or `application/json` in some cases) for the body. The CNC API expects YANG data in JSON format, which is indicated by `application/yang-data+json` in the swagger docs [65]. It's generally safe to use that as Content-Type for the POST body.

In Postman, you might do the following: - Create a **"Get TGT"** request with the username/password. On success, save the response (the TGT string) to an environment variable (Postman has scripting for tests to capture response values). - Create a **"Get JWT"** request which uses the TGT variable in its body. Save the returned JWT to an environment variable (e.g., `{{jwt_token}}`). - For all other requests (like **"Create RSVP Tunnel"**, **"Create SR Policy"**), set an `Authorization` header with value `Bearer {{jwt_token}}`. Also set `Content-Type: application/yang-data+json` and body as needed. You can also set the base URL and port as variables if you like, to avoid repetition.

Important: The JWT token by default is valid for 8 hours [66] [67]. After that, you'll need to repeat the authentication process to get a new token. (If you get HTTP 401 Unauthorized on API calls and it's been >8 hours, likely the token expired.)

Make sure your CNC's HTTPS certificate is handled properly. If it's self-signed (common in lab setups), you might need to disable SSL verification in Postman or import the cert. In curl examples above, `-k` was used to ignore certificate validation for simplicity [68] [69].

Example Postman usage for creating an SR Policy: 1. Do the login sequence to set the token. 2. Construct a POST request: `https://10.0.0.1:30603/crosswork/nbi/optimization/v3/restconf/operations/cisco-crosswork-optimization-engine-sr-policy-operations:sr-policy-create` 3. Headers: - `Authorization: Bearer <your JWT>`, - `Content-Type: application/yang-data+json`. 4. Body: raw JSON as in the examples above (e.g., specify head-end IPs, color, etc.). 5. Send the request. You should get a 200 OK and a JSON response with state. You can check the body or the Postman "Pretty" view to see the `state: success` and any message.

If the response has `state: failure`, you might have an error. Common errors could be: - Invalid device IP (head-end not known or not onboarded in CNC). - PCE not able to compute a path (no route meeting constraints). - Malformed payload (JSON error or missing required field). - Not specifying `api_key`/token correctly – ensure the header is exactly `Authorization` (Postman sometimes also has an "Auth" tab where you can just input the token as Bearer token).

By using Postman collections or environments, you can streamline this. Cisco often provides a Postman collection or example environment for CNC APIs (in Cisco DevNet or documentation).

Automation with Python: End-to-End Script Example

For integration into scripts or tooling, Python can be used (with the `requests` library, for instance) to automate the process of authentication and tunnel provisioning. Below is a simplified example of how a Python script might perform these steps:

```
import requests

CNC_HOST = "10.0.0.1"
CNC_PORT = 30603
USERNAME = "admin"
PASSWORD = "YourPassword123"

# Disable warnings for self-signed certs (if needed)
requests.packages.urllib3.disable_warnings()

# Step 1: Get TGT
tgt_url = f"https://{CNC_HOST}:{CNC_PORT}/crosswork/sso/v1/tickets"
tgt_resp = requests.post(tgt_url, data={
    "username": USERNAME,
    "password": PASSWORD
}, headers={"Content-Type": "application/x-www-form-urlencoded", "Accept":
"text/plain"}, verify=False)
if tgt_resp.status_code != 200:
    raise Exception(f"Failed to get TGT: {tgt_resp.status_code}
{tgt_resp.text}")
tgt = tgt_resp.text
print("Obtained TGT:", tgt)

# Step 2: Get JWT using the TGT
jwt_url = f"https://{CNC_HOST}:{CNC_PORT}/crosswork/sso/v2/tickets/jwt"
jwt_resp = requests.post(jwt_url, data={
    "tgt": tgt,
    "service": f"https://{CNC_HOST}:{CNC_PORT}/app-dashboard"
}, headers={"Content-Type": "application/x-www-form-urlencoded"}, verify=False)
if jwt_resp.status_code != 200:
    raise Exception(f"Failed to get JWT: {jwt_resp.status_code}
```

```
{jwt_resp.text}")
jwt_token = jwt_resp.text
print("Obtained JWT token (truncated):", jwt_token[:50], "...")

# Step 3: Use JWT to create a tunnel (example for RSVP-TE)
api_url = f"https://{CNC_HOST}:{CNC_PORT}/crosswork/nbi/optimization/v3/
restconf/operations/cisco-crosswork-optimization-engine-rsvp-te-tunnel-
operations:rsvp-te-tunnel-create"
headers = {
    "Authorization": f"Bearer {jwt_token}",
    "Content-Type": "application/yang-data+json"
}
payload = {
    "input": {
        "rsvp-te-tunnels": [
            {
                "head-end": "192.0.2.1",
                "end-point": "192.0.2.5",
                "path-name": "LSP_A_TO_B",
                "rsvp-te-tunnel-path": {
                    "optimization-objective": "te-metric"
                }
            }
        ]
    }
}
resp = requests.post(api_url, json=payload, headers=headers, verify=False)
print("Tunnel create status:", resp.status_code)
print("Response:", resp.json())
```

In this script: - We post to `/sso/v1/tickets` with credentials to get `tgt`. - Then post to `/sso/v2/tickets/jwt` to get the `jwt_token`. - Then we prepare a payload for an RSVP-TE tunnel (from 192.0.2.1 to 192.0.2.5) with minimal constraints (just shortest TE path). - We call the create API with the JWT in the header.

We print out the result. For example, if successful, `resp.json()` might be:

```
{'output': {'results': [{'head-end': '192.0.2.1', 'end-point': '192.0.2.5',
'path-name': 'LSP_A_TO_B', 'message': 'Tunnel created', 'state': 'success'}]}}
```

We could further parse this to verify `state` is success. We could also perform a GET operation to retrieve the details of the created tunnel for validation.

Validating Outcomes: There are multiple ways to validate that the tunnels/policies were created successfully: - **Check API Response:** As above, the immediate API response tells you if the operation was successful. A `success` state generally means the PCE has instantiated the tunnel. If `failure`, examine

the `message` for why. - **Retrieve Tunnel Info via API:** CNC likely has GET endpoints for fetching current TE tunnels or policies. For instance, an operational GET might be at `.../restconf/data/cisco-crosswork-optimization-engine-rsvp-te-tunnel...` to list tunnels, or an SR policy oper data endpoint. (The DevNet docs mention data models like `data_cisco-crosswork-segment-routing-policy_sr` etc.) One could call those to confirm the tunnel appears and see its path. For brevity, we skip detailed GET examples. - **CNC UI:** Logging into the CNC web UI, you can navigate to **Services & Traffic Engineering > Traffic Engineering** and then check the RSVP-TE or SR-MPLS/SRv6 tables for the new entry [70] [71]. The tunnel should be listed (with state Up if established). The topology map can also highlight the path if you select the policy [72] [73]. - **Device CLI:** Ultimately, the head-end router will have a new tunnel interface or SR policy. For example, on IOS-XR, you could check `show mpls traffic-eng tunnels brief` for RSVP, or `show segment-routing traffic-eng policy` for SR. You would see the tunnel/policy created by PCE (often with a identifier showing it's PCE-initiated). Since the question focuses on external interfaces, an API user may not SSH into devices, but for completeness this is another verification.

Using the Python script approach, one can integrate CNC control into orchestration tools or CI/CD pipelines. For instance, you could have a script that reads desired policies from a file and uses the API to ensure those exist in the network, or one that reacts to network events (congestion triggers) by calling CNC APIs.

Finally, let's illustrate a sample use-case topology and how these pieces come together.

Example Use-Case Topology and Workflow

Figure 2: Example Network Topology for TE Policies. In this sample, PE-A, PE-B, and PE-C are edge routers, and P-TOPLEFT/RIGHT and P-BOTTOMLEFT/RIGHT are core routers. Multiple links (dashed lines) connect the nodes. This network can support various TE tunnels – e.g., an SR policy from PE-A to PE-B might traverse the top path or bottom path. The SR-PCE (not shown) has visibility of all links and metrics to compute optimal routes. [74]

Imagine the above network: a small ISP core with 4 P routers forming a square, and 3 PE routers on the edges. Let's say: - PE-A (left), PE-B (top-right), PE-C (bottom-right) are provider edges. - We want to establish an engineered tunnel from PE-A to PE-B (perhaps to carry a high-priority service). We'll use CNC's API to create an SR-MPLS policy for this.

Use-Case: PE-A to PE-B low-latency SR Policy with FRR. Suppose the default IGP path from PE-A to PE-B goes through P-BOTTOMLEFT -> P-BOTTOMRIGHT -> P-TOPRIGHT (a longer path with more hops), but there is a slightly risky shortcut via P-TOPLEFT -> P-TOPRIGHT that has fewer hops but one link is of lower capacity. We want the lowest latency path, but only if links are protected.

Steps to deploy via CNC API: 1. **Choose head-end and endpoint:** Head-end = PE-A's loopback (e.g., 100.100.100.5), Endpoint = PE-B's loopback (100.100.100.6) [75]. 2. **Determine constraints:** We want low latency (`optimization-objective: delay`) and only use protected links (`protected: true`). Assume all core links have backup except one – the P-BOTTOM path has unprotected segment. The PCE will thus prefer the top path if it's protected. 3. **Call the API:** We use `sr-policy-create` with head-end=100.100.100.5, end-point=100.100.100.6, color say 77, and in sr-policy-path we set `"path-optimization-objective": "delay", "protected": true`. 4. **Result:** The PCE computes the path. Let's say it picks PE-A -> P-TOPLEFT -> P-TOPRIGHT -> PE-B as that is the lowest delay and all those links are

protected by TI-LFA. It installs the SR policy on PE-A with the SID list [P-TOPLEFT Node-SID, P-TOPRIGHT Node-SID, PE-B Node-SID]. CNC returns success. 5. **Validation:** The response shows success. In the CNC UI, under SR-MPLS policies, we see a new entry for headend PE-A to endpoint PE-B with Color 77, and status UP. If we click on it, we can view details like the actual path (segment list) and metrics [76] [77]. The delay is shown (via SR-PM) on that policy card, confirming it took the low-latency route [78]. 6. If we had a requirement for a secondary path, we could similarly create another candidate (maybe color 78 or same color with different path-name if supported) with a disjointness constraint from the primary.

In the topology figure, the dashed green lines show connectivity. If one of those links failed, because we chose protected links, local repair on the routers would kick in (TI-LFA immediately provides a detour). CNC would also be monitoring via telemetry and could recompute if needed (it could signal a re-optimization if a link metric changed significantly, for example).

For an RSVP-TE example, say between PE-A and PE-C, we might require 1 Gbps bandwidth. Using the RSVP API with `signaled-bandwidth`, the PCE could try to find a path with available capacity. If the direct path is full, it might route it longer way. CNC (via telemetry) would update traffic collector stats and we could see the reserved bandwidth on each link [79].

This end-to-end scenario shows how an external OSS/BSS or script can program the network through CNC's NB APIs, leveraging the SR-PCE intelligence without manually configuring each device. The operator specifies **intent (endpoint, constraints)** and CNC/PCE handle the rest (finding the path, signaling it, and tracking it).

Conclusion

Segment Routing PCE in Cisco Crosswork Network Controller 7.1 enables powerful traffic engineering capabilities through a set of well-defined northbound APIs. We covered the architecture – where an IOS-XR-based PCE works in tandem with CNC – and how it supports RSVP-TE, SR-MPLS, and SRv6 tunnels. The NB APIs provide a model-driven way to create TE tunnels with various constraints like affinities, disjointness groups, bandwidth and latency optimization, and more. We demonstrated the usage of these APIs including authentication (JWT-based) and provided examples for both Postman and Python automation. By using these interfaces, network operators or applications can perform intent-based provisioning of traffic engineered paths in a multi-vendor, multi-protocol network, without needing to touch individual router configs. All interactions are via RESTCONF APIs with JSON payloads, making integration into higher-level orchestration systems or OSS straightforward.

Through this approach, Cisco CNC abstracts the complexity of path computation and signaling, allowing external systems to request **"Tune the network like this"** and the controller takes care of **"how to implement it"**. This aligns with modern network automation principles and enables use-cases from on-demand bandwidth steering to fast failure recovery and SLA assurance for premium services.

Sources:

- Cisco DevNet – Crosswork Network Controller 7.1 API Documentation [59] [60] [80] [64]
- Cisco Crosswork Network Controller 7.1 Installation and Administration Guides [81] [8]
- Cisco Crosswork Network Controller 7.1 Release Notes [1] (SR-PCE description)

- Cisco Crosswork Network Controller 7.0 Solution Workflow Guide [2] [82] (architecture and PCE integration)
- Cisco Crosswork Traffic Engineering and Optimization Guide [10] [11] (SR-MPLS/SRv6 UI usage)
- API Swagger extracts for RSVP-TE and SR policy create operations [83] [35] [22] [36].

---

[1] [9] [53] Release Notes for Cisco Crosswork Network Controller, Release 7.1.0 - Cisco
https://www.cisco.com/c/en/us/td/docs/cloud-systems-management/crosswork-network-controller/7-1/Release-Notes/release-notes-for-cisco-crosswork-network-controller-release-7-1-0.html

[2] [3] [5] [6] [82] Cisco Crosswork Network Controller 7.0 Solution Workflow Guide - Solution Overview [Cisco Crosswork Network Automation] - Cisco
https://www.cisco.com/c/en/us/td/docs/cloud-systems-management/crosswork-network-controller/7-0/Solution-Workflow-Guide/bk-crosswork-network-controller-7-0-solution-workflow-guide/m-solution-overview.html

[4] [7] [8] [79] [81] Cisco Crosswork Network Controller 7.1 Installation Guide - Integrate SR-PCE [Cisco Crosswork Network Automation] - Cisco
https://www.cisco.com/c/en/us/td/docs/cloud-systems-management/crosswork-infrastructure/7-1/InstallGuide/b_cisco_crosswork_7_1_install_guide/m_cw-5-0-integrate-srpce.html

[10] [11] [12] [13] [14] [48] [49] [57] [70] [71] [72] [73] [76] [77] [78] Cisco Crosswork Network Controller 7.1 Traffic Engineering and Optimization - SR-MPLS and SRv6 [Cisco Crosswork Network Automation] - Cisco
https://www.cisco.com/c/en/us/td/docs/cloud-systems-management/crosswork-network-controller/7-1/CNC-Traffic-Engineering/b-cnc-traffic-engineering-7-1/m2-sr-mpls-and-srv6-.html

[15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [65] [83] Create an RSVP-TE tunnel - Crosswork Network Controller 7.1 APIs - Cisco DevNet
https://developer.cisco.com/docs/crosswork/network-controller/create-an-rsvp-te-tunnel/

[31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [44] [45] [46] [50] [51] Create an SR policy - Crosswork Network Controller 7.1 APIs - Cisco DevNet
https://developer.cisco.com/docs/crosswork/network-controller/create-an-sr-policy/

[47] Overview - Crosswork Network Controller 7.1 APIs - Cisco DevNet
https://developer.cisco.com/docs/crosswork/network-controller/segment-routing-policy-operations-api-overview

[52] [PDF] CrossWork APIs Zero to Hero - Cisco Live
https://www.ciscolive.com/c/dam/r/ciscolive/emea/docs/2023/pdf/DEVNET-2405.pdf

[54] [55] [56] [58] Please read
https://www.ciscolive.com/c/dam/r/ciscolive/apjc/docs/2024/pdf/BRKSPG-2551.pdf

[59] Introduction - Crosswork Network Controller 7.1 APIs - Cisco DevNet
https://developer.cisco.com/docs/crosswork/network-controller/

[60] [61] [62] [63] [64] [67] [68] [69] [80] Authentication - Crosswork Planning 7.1 API - Cisco DevNet
https://developer.cisco.com/docs/crosswork/planning/authentication/

[66] [74] [75] Getting Started - Crosswork Network Controller 7.1 APIs - Cisco DevNet
https://developer.cisco.com/docs/crosswork/network-controller/intent-based-service-provisioning-getting-started/