Network Service Mesh Dashboard specification

[**Open Questions 1**](#_x0wg6p7irar)

[**Overview 2**](#_owgv2k4dc5cr)

[Problem 2](#_outxze3865aw)

[Goals 2](#_j3zmkv7majgq)

[User Acceptance Criteria (Requirements) 2](#_bdyk61o68ebs)

[Solution limitations 3](#_k1yfw9v4p6a1)

[Use cases 3](#_vo66wq65iu4v)

[**UI Pages 5**](#_r3p3ux38unpb)

[Overview 6](#_z7nhtbe20apl)

[Dataplane view 6](#_auto6pw69yte)

[vL3 + remote configuration mockup 6](#_j9t59xg9ftyi)

[View component details 7](#_d987x2utfzet)

[Controlplane view 7](#_dfa7mnvbg8nq)

[vl3 configuration with NetworkService-links 7](#_k7dghvqzpzkf)

[vl3 configuration with registry-links 8](#_fj5jl3i0r79s)

[Services view 9](#_dqo1eoflsfpx)

[Overview view 9](#_co34zi4vo7yo)

[**MVP description 9**](#_59bon9a9zz7t)

[**Implementation Description 9**](#_lmzubksxjy27)

[High-level Architecture 10](#_ahsmhlgaj5a1)

[Methodology Overview 10](#_vlbrrj8znakw)

[Data models 11](#_e0d4blohxb1x)

[UI Functionality 13](#_gnycggmmkwo7)

[API 14](#_mfgqa0awb3pp)

[Frontend 14](#_nu6lvior6ekh)

[Development stack 14](#_l4n10x57tade)

[**Team 14**](#_b6n08twh40qf)

[**Appendix A. Glossary 15**](#_44z6qg59l4ap)

[**Appendix B. Data models 16**](#_3x3ehw4q8hjr)

**Development Project Links:**

<https://github.com/bellycat77/dashboard>

# Open Questions

1. Which architecture type should we use:
   1. Frontend -> Backend -> NSM
   2. Frontend -> Service (data collector) -> NSM (preferable)
2. Will we send requests to NSM registry, or to nsmgr only?

# Overview

## Problem

1. Sometimes, it is difficult for the DevOps-engineer to identify the topology of the network topology provided by NSM. There is no tool for topology visualization at this time.

2. There are no examples of applications to visualize network service mesh internals at this moment. If it would be helpful to have such an application. And probably, it might to make NSM more popular

## Goals

The goal of this project is to create a UI to display the topology of the Network Service Mesh in various aspects, displaying the state of components and operating statistics:

* Dataplane view
* Controlplane view
* view of services and their clients
* display statistics

The solution should be an example of using the NSM client API (to be open source and documenting some aspects of the solution regarding the use of the API)

## User Acceptance Criteria (Requirements)

1. The dashboard should visually display the topology of the NSM in various views:
   1. Dataplane
   2. Controlplane
   3. Services
2. In the case of Dataplane, the dashboard should be able to display both the original NSM topology and a data-flow view (topology decomposed by connections)
3. For the case of a large topology, the dashboard should be able to show an aggregated view of this topology
4. In the case of an aggregated topology view, drill down to the selected element with a mouse click
5. The dashboard should display additional information about the state of the NSM, in the form of indicators, sensors, tables, and graphs:
   1. volume of transmitted/received data
   2. error statistics
   3. availability/operability of individual clients/endpoints
   4. statistics on services (number of endpoints per service, their identifiers, distribution among cluster nodes)
   5. for interdomain case, nodes distribution among clusters

## Solution limitations

1. The solution should use Network Service Mesh API
2. Dashboard should be a web application
3. NSM configurations Local, Remote, vL3, vL3+Remote are supported
4. The dashboard displays the workloads on the same node it is working on.
5. The number of PODs on a node should not exceed the recommended number of PODs <https://kubernetes.io/docs/setup/best-practices/cluster-large/>
6. To reduce cost and development time, the dashboard should use existing 3-rd party solutions/components to display topologies
7. Graph rendering performance and layouts depend on the graph visualization library used
8. The solution will be developed with an iterative approach

## Use cases

| Use case name | Overview page |
| --- | --- |
| Use case number | 1 |
| Use case goal | User needs to see common statistics of NSM |
| Primary actor | User |
| Preconditions | Dashboard application is opened in browser |
| Basic flow | User clicks on “Overview” tab. The Dashboard shows the Overview view with common statistics of NSM |
| Alternative flows | When getting information from NSM the error has occurred. The Dashboard displays the error details instead of the Overview page. |

| Use case name | Dataplane/Topology page |
| --- | --- |
| Use case number | 2 |
| Use case goal | User needs to see topology of NSM Dataplane |
| Primary actor | User |
| Preconditions | Dashboard application is opened in browser |
| Basic flow | User clicks on “Dataplane/Topology” tab. The Dashboard shows Dataplane/Topology view |
| Alternative flows | When getting information from NSM the error has occurred. The Dashboard displays the error details instead of the Dataplane/Topology page. |

| Use case name | Dataplane/Connections page |
| --- | --- |
| Use case number | 3 |
| Use case goal | User needs to see connections-related topology of NSM Dataplane |
| Primary actor | User |
| Preconditions | Dashboard application is opened in browser |
| Basic flow | User clicks on “Dataplane/Connections” tab. The Dashboard shows Dataplane/Connections view |
| Alternative flows | When getting information from NSM the error has occurred. The Dashboard displays the error details instead of the Dataplane/Connections page. |

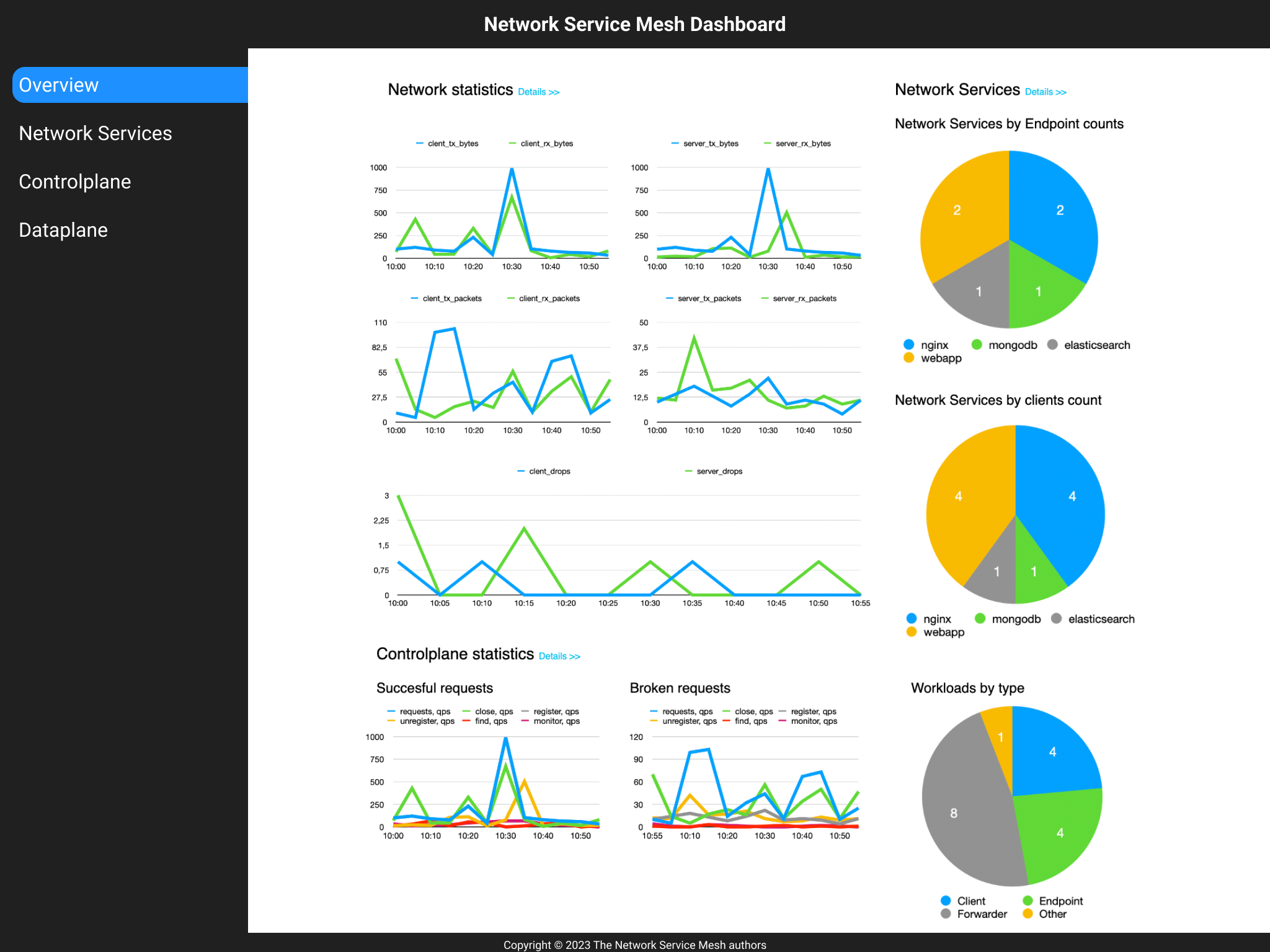
| Use case name | Controlplane page |
| --- | --- |
| Use case number | 4 |
| Use case goal | User needs to see the topology of NSM Controlplane |
| Primary actor | User |
| Preconditions | Dashboard application is opened in browser |
| Basic flow | User clicks on “Controlplane” tab. The Dashboard shows the Controlplane view |
| Alternative flows | When getting information from NSM the error has occurred. The Dashboard displays the error details instead of the Controlplane page. |

| Use case name | Services page |
| --- | --- |
| Use case number | 5 |
| Use case goal | User needs to see the services related topology of NSM |
| Primary actor | User |
| Preconditions | Dashboard application is opened in browser |
| Basic flow | User clicks on “Services” tab. The Dashboard shows the Services view |
| Alternative flows | When getting information from NSM the error has occurred. The Dashboard displays the error details instead of the Services page. |

| Use case name | View control |
| --- | --- |
| Use case number | 6 |
| Use case goal | User needs to manipulate Dataplane/Controlplane/Services topology of NSM |
| Primary actor | User |
| Preconditions | Dashboard application is opened in browser on page Dataplane, Controlpane, or Services |
| Basic flow | User can zoom in, zoom out, or zoom to fit view by using View control. Also, user can show/hide Legend for topology. |

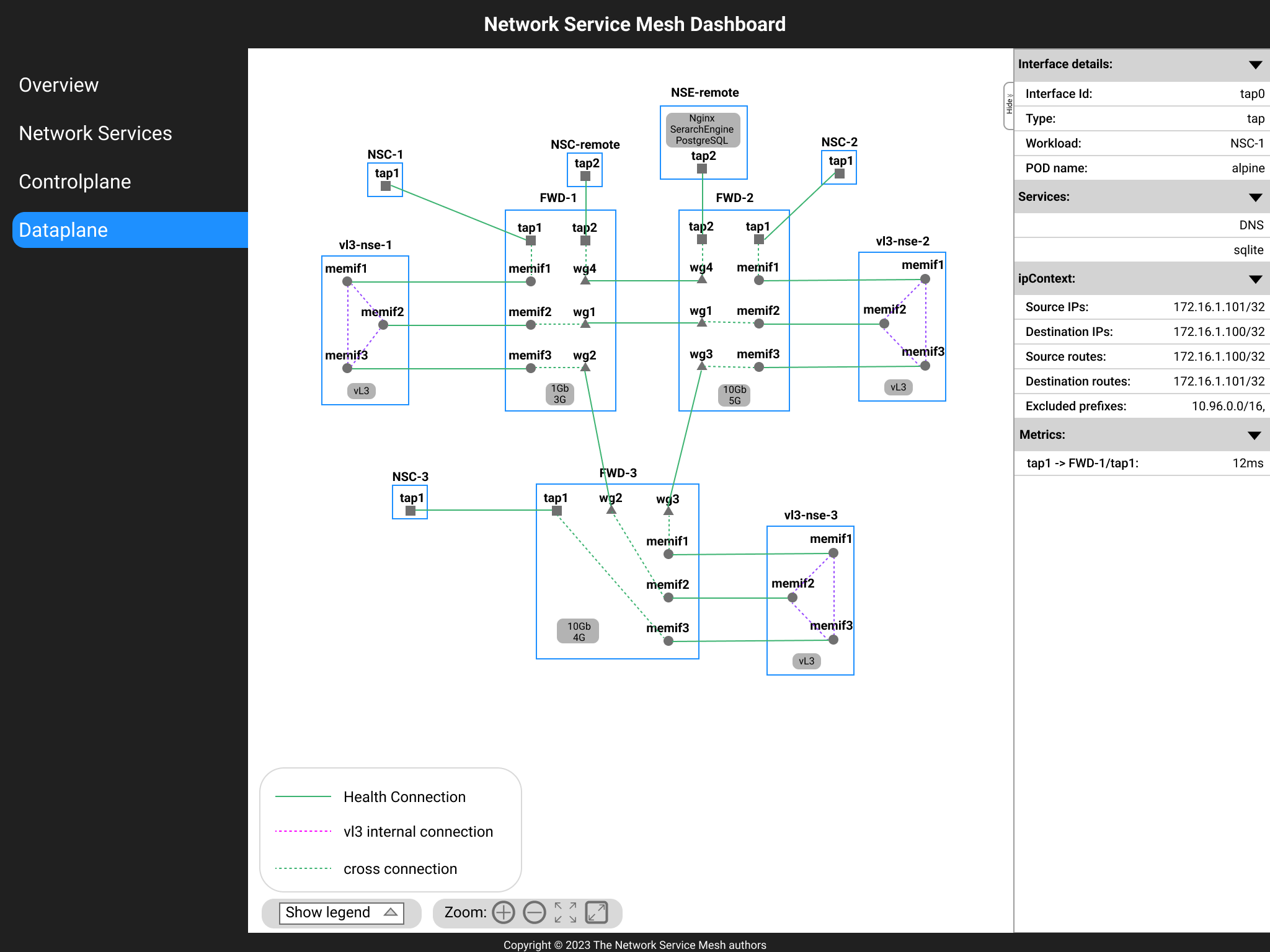
# UI Pages

### Overview



### Dataplane view

#### vL3 + remote configuration mockup



#### View component details

The dataplane view should display dataplane’s components with interfaces and connections between them. Healthy connections should have a green color, and broken connections should be red colored.

Сlicking on an element should open a panel with detailed information for that element. The panel should contain additional information like routes, IP-addresses, masks, **<To be done>**

### Controlplane view

#### vl3 configuration with NetworkService-links

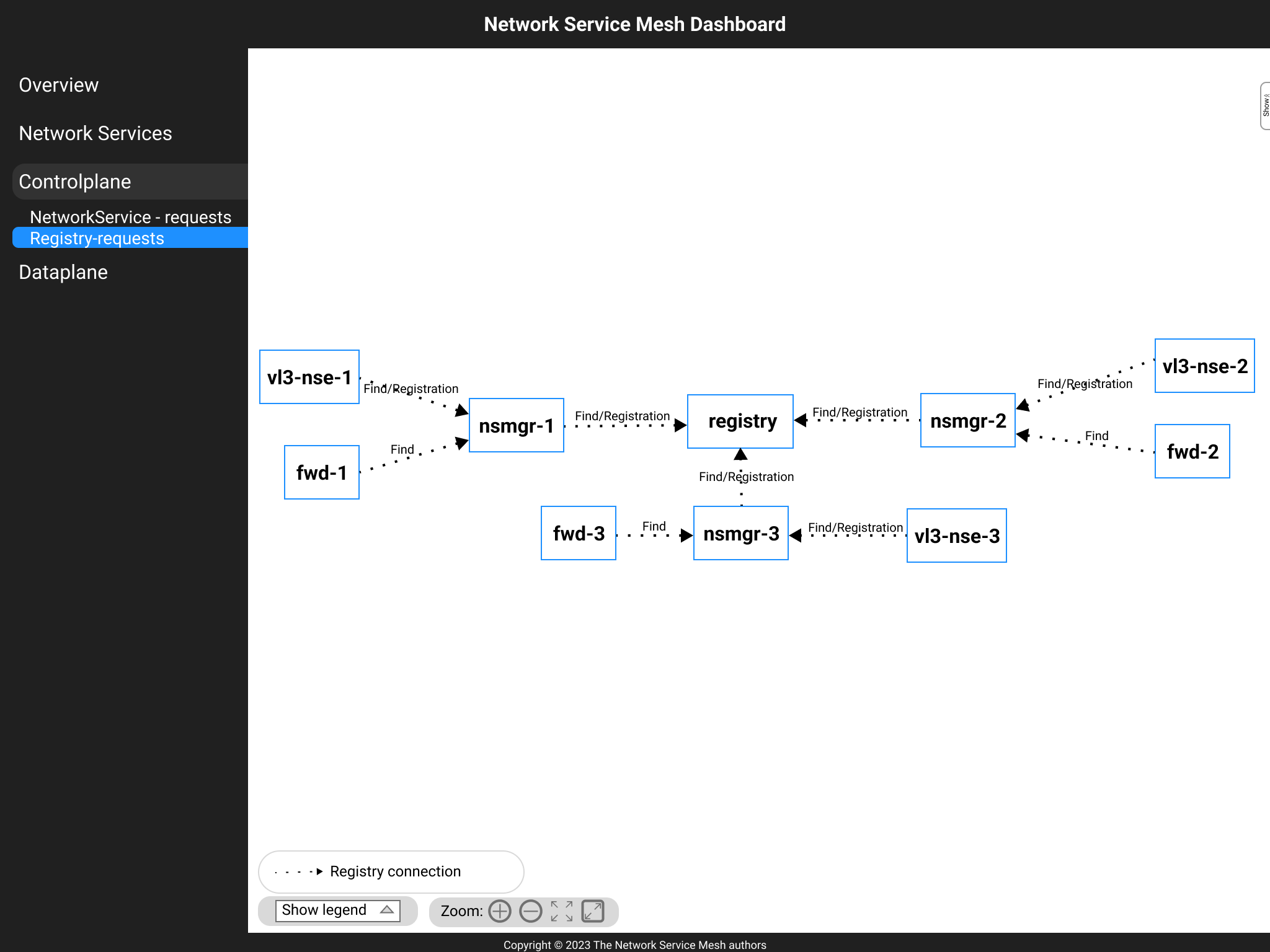
On this view, links with requests for network services are presented only

### 

#### 

#### vl3 configuration with registry-links

On this view, links with requests for registry services are presented only



### 

### Services view

**<To be done>**

### Overview view

**<To be done>**

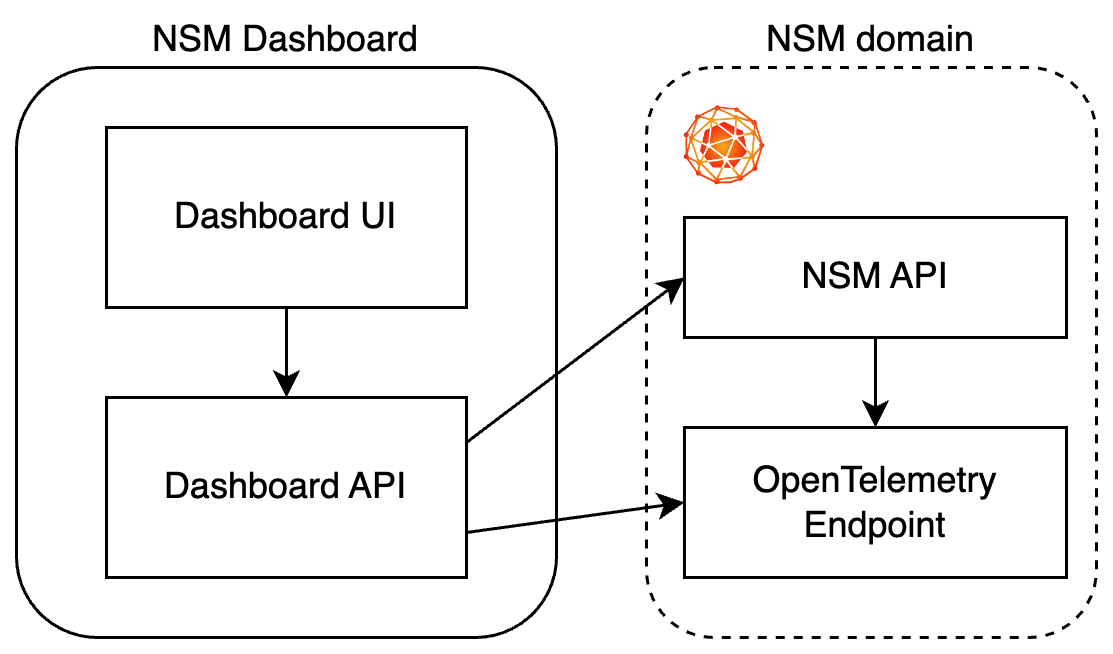
# MVP description

Functionality:

* contains Dataplane view only
* automatic elements layout on view
* displays only endpoints interacting with dashboard’s node where Dashboard backend works

# Implementation Description

## High-level Architecture

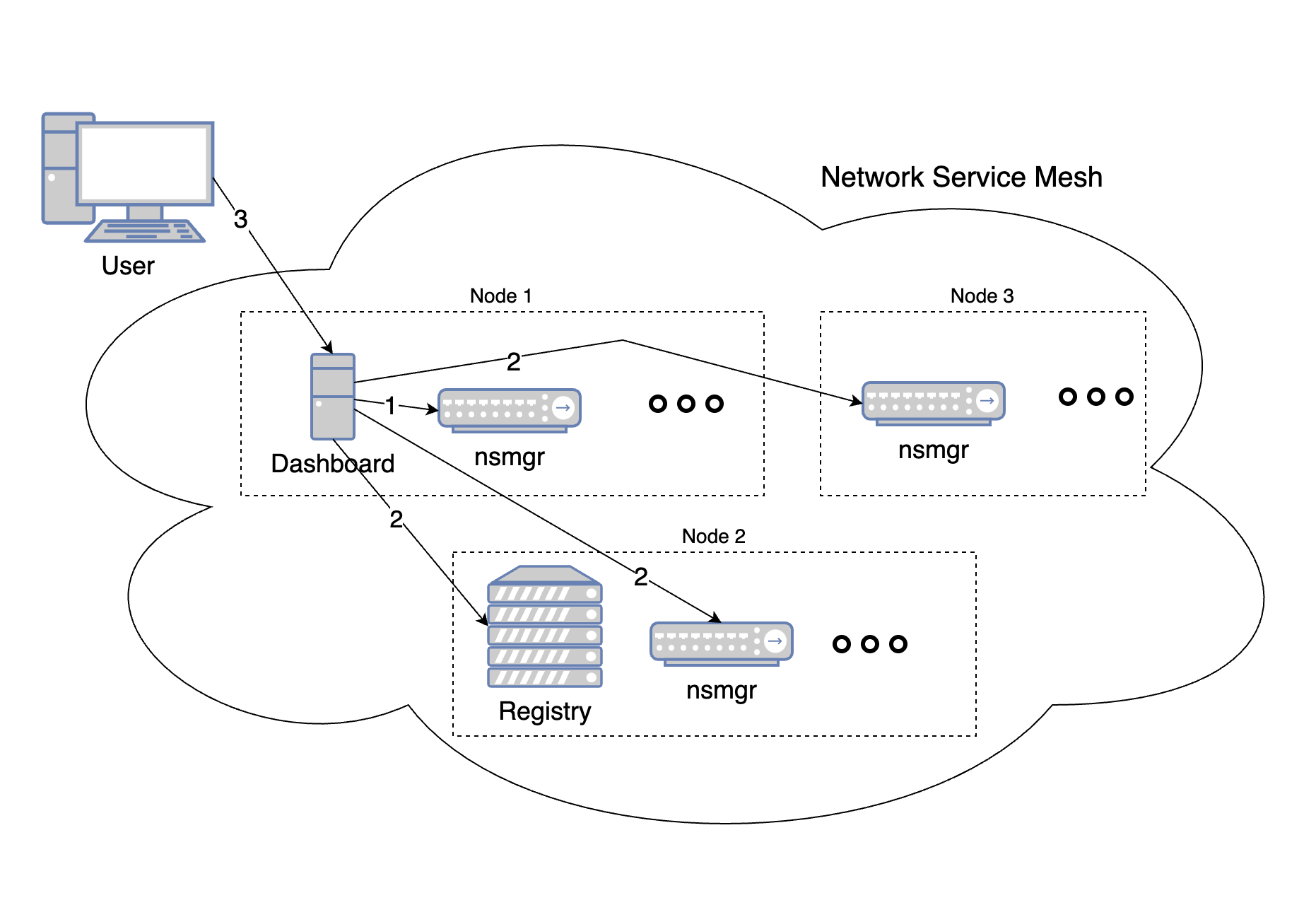


## Methodology Overview

Proposed methodology for obtaining, processing, and storing data:

1. All displayed data is processed and stored on the client side (in memory and/or local browser storage)
2. When opening the NSM Dashboard web interface, the user can access only the data stored in the browser's local storage, if their storage period has not expired.
3. The initial page of the application is the Overview view.
4. The data is displayed on all views as the data is loaded.
5. A separate module (Backend) is responsible for loading and processing NSM state data. When the application starts, this module connects to nsmgr running on the same node as the Dashboard itself. The module receives nsmg data, processes, and retrieves connection data for this nsmgr. Next, the module iteratively makes queries to other nsmgrs ***and to the registry(?)***, extracting information about the state of the components presented in the NSM. Data collection is ongoing. Each new data collection cycle begins immediately after the end of the previous cycle and/or a specified time interval has elapsed since the start of the previous cycle.
6. The interconnection module (a part of Frontend) periodically requests the processed data from the Backend and stores it in in-memory storage (State) and in the browser's local storage. This module also handles connection/data reception errors and stores error information in in-memory storage. Error information is displayed as a status bar in the browser, regardless of the active data view. This module also invalidates data after a specified storage period has expired. If information about the layout of elements in views is stored in the browser's local storage, it is invalidated when the set of NSM elements changes
7. Data views display data stored in in-memory storage and in the browser's local storage. The Services, Dataplane, and Controlplane views allow users to change the layout of node graph elements. Information about the customized layout of elements is saved in the browser's local storage. When refreshing a page or switching views, it first checks for saved layout data. If data is available, it is used. If there is no data, then an automatic layout is used. If there is no data for some reason, the views should display a related message.
8. The Services, Dataplane, and Controlplane views must be able to limit the amount of data they display.
9. The user can switch between views without losing accumulated data. If the user has changed the layout in the view, then these changes are saved until the cluster configuration is changed or it is rebooted

Data flows:



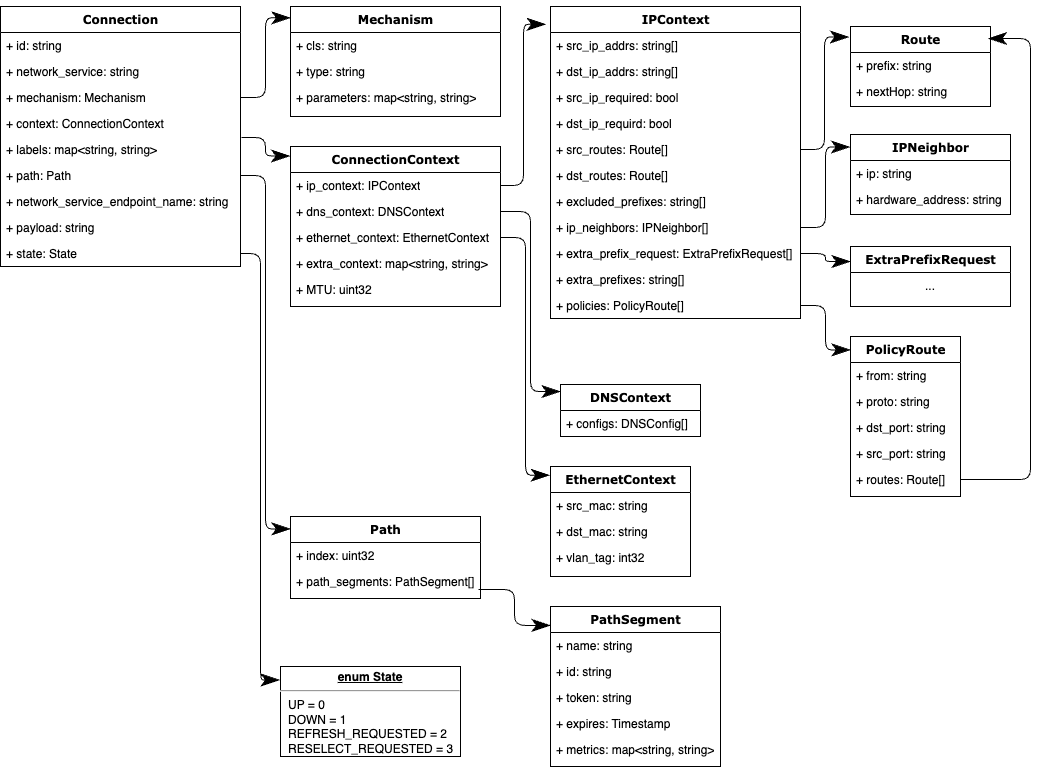
1 - requesting data by dashboard backend from nsmgr in the same node

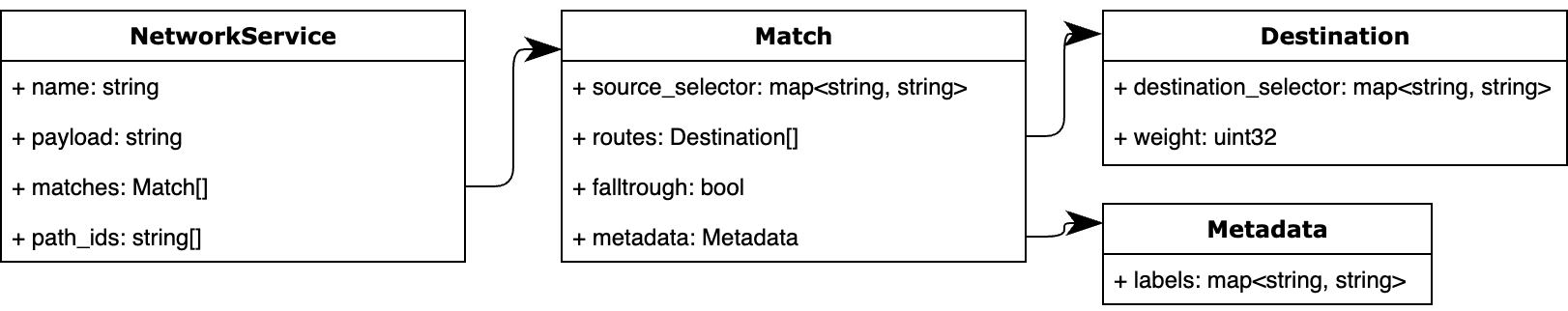
2 - requesting data by dashboard backend from other nsmgrs in Service Mesh and to Registry

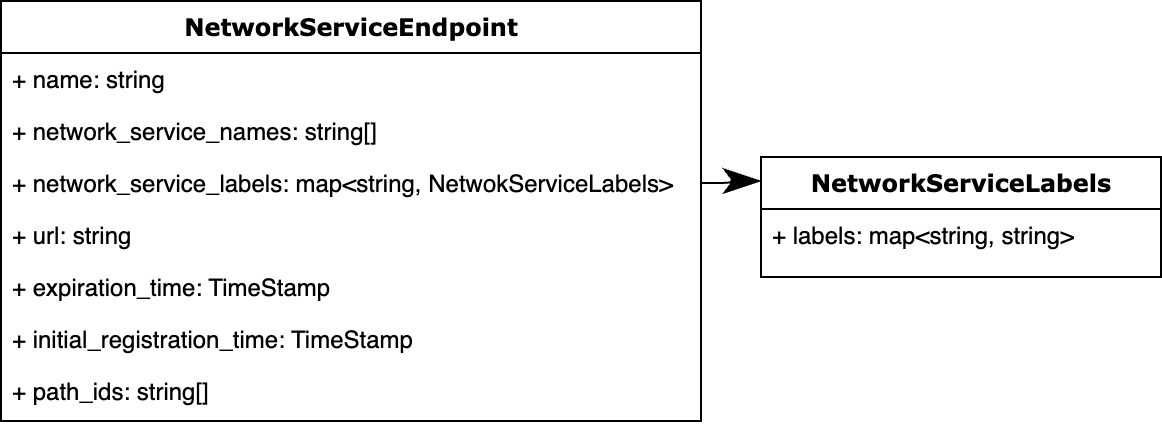
3 - requesting data by dashboard frontend from dashboard backend

## Data models

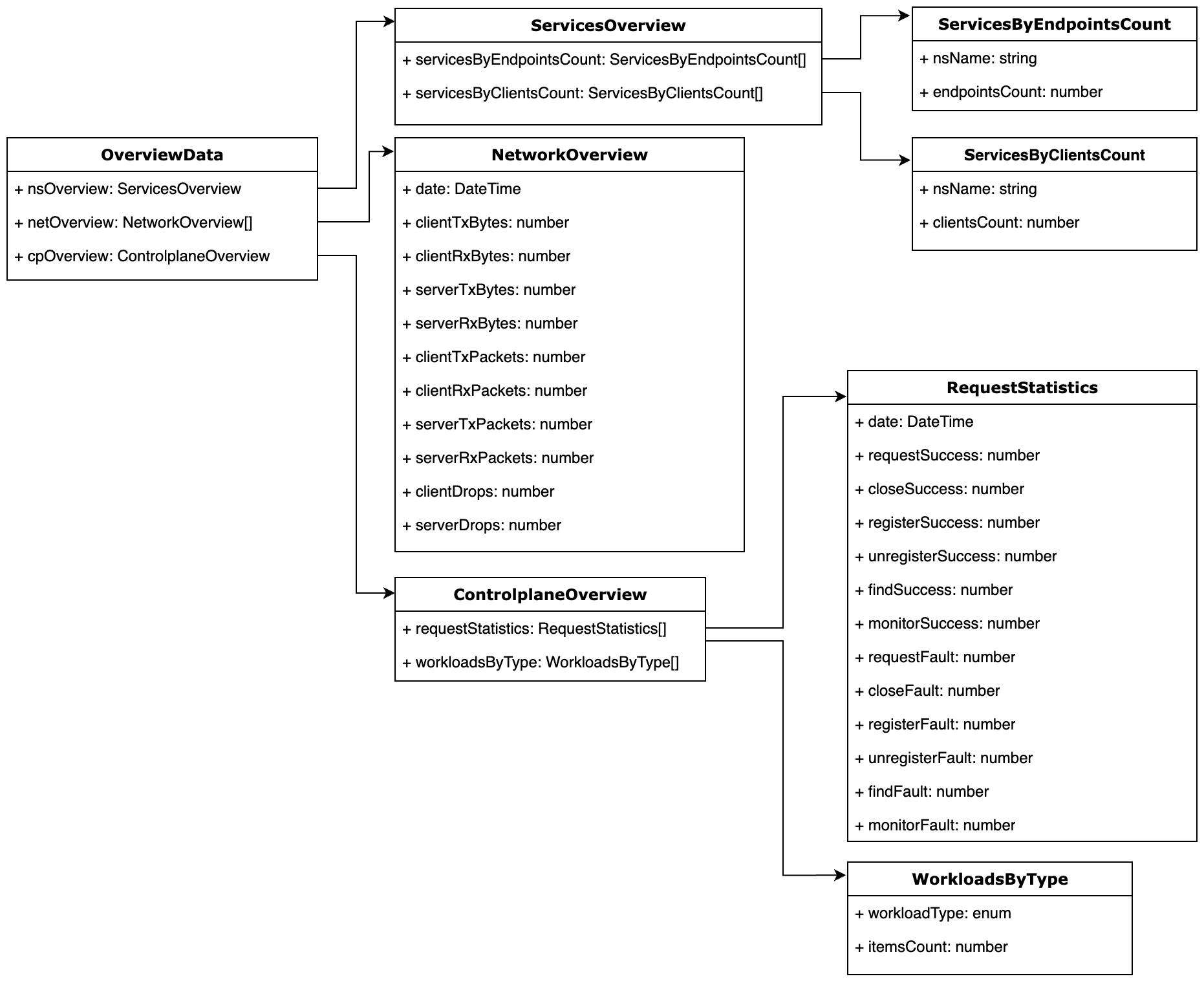
Models of data used in the project by backend:

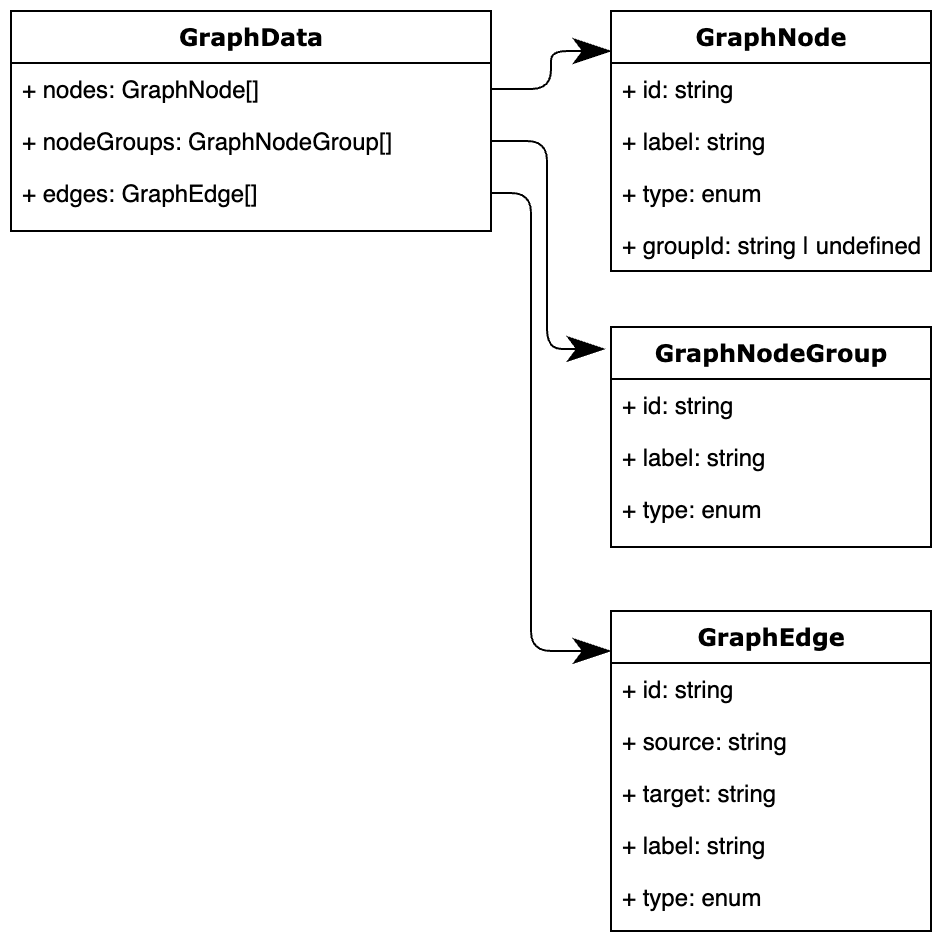


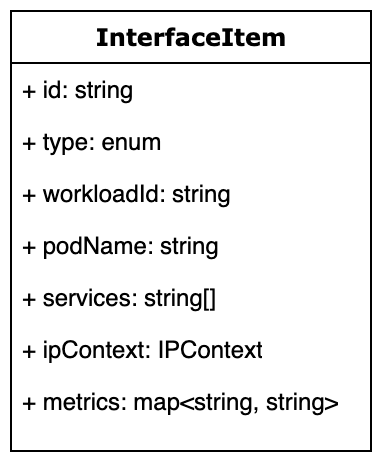
****

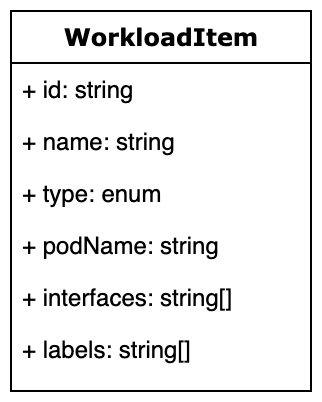
****

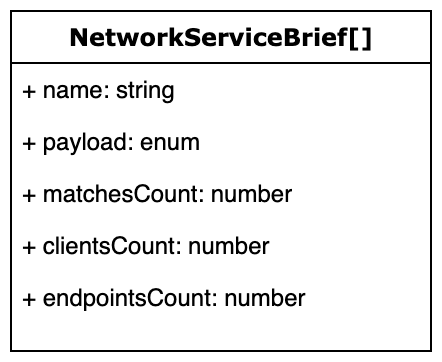
The view data models:

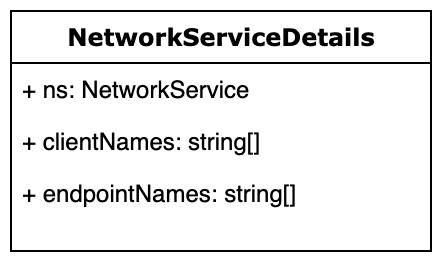












## UI Functionality

1. The Overview view displays: **<to be done>**
2. The Services, Dataplane, and Controlplane views display data stored in in-memory storage and in the browser's local storage. The Services, Dataplane, and Controlplane views allow users to change the layout of graph elements. Information about the customized layout of elements is saved in the browser's local storage. When refreshing a page or switching views, it first checks for saved layout data. If data is available, it is used. If there is no data, then an automatic layout is used. If there is no data for some reason, the views should display a related message.
3. The Services, Dataplane, and Controlplane views must be able to limit the amount of data they display. For example, the user should be able to see only those nodes and edges that have a direct or indirect connection to the node he selects.
4. It should be possible to switch between views according to the following rules: <To be done>

## API

Network Service Mesh GRPC API [link](https://github.com/networkservicemesh/api)

### Frontend

Frontend should work as a part of NSM inside the cluster. It should collect data about connections from NSM managers. The collected data should be processed to a set of nodes/edges for rendering by the visualization framework in Dataplane and Controlplane views.

Switch between views should be made by selecting a left-side menu item.

The user should have the ability to move elements of the view. The view should preserve the elements' layout in case of browser window reload or in case of switching between views.

### Development stack

* Node.js (JavaScript runtime environment [link](https://nodejs.org/en))
* React.js (the library for web user interfaces [link](https://react.dev/))
* Cytoscape.js (graph theory (network) library [link](https://js.cytoscape.org/))
* grpc-web (gRPC-Web Client Runtime Library [link](https://www.npmjs.com/package/grpc-web))
* Typescript (programming language [link](https://www.typescriptlang.org/))

# Team

Developer: Ivan Osipov <[ivan.osipov@xored.com](mailto:ivan.osipov@xored.com)>

Reviewers: Denis Tingaikin, Daniil Efremov

UI/UX Designer: Alexander Lapin

# Appendix A. Glossary

**NSM configuration Local** - configuration when client and endpoints are connected to the same forwarder

**NSM configuration Remote** - configuration when client and endpoints belong to the same domain, but are connected to different forwarders

**NSM configuration vL3** - all workloads in domain are connected to each other using vl3-endpoints

**NSM configuration vL3** **+ remote** - the combination of vL3 and Remote configurations, some clients are connected using vl3-endpoints, some clients not use vl3-endpoints

# Appendix B. Data models

Data models used by backend / data collector service

**Connection:**

{

string id;

string network\_service;

Mechanism mechanism;

ConnectionContext context;

map<string, string> labels;

Path path;

string network\_service\_endpoint\_name;

string payload;

State state;

}



**Mechanism:**

{

string cls;

string type;

map<string, string> parameters;

}



**ConnectionContext**:

{

IPContext ip\_context;

DNSContext dns\_context;

EthernetContext ethernet\_context;

map<string, string> extra\_context;

uint32 MTU;

}



**IPContext:**

{

string src\_ip\_addrs[];

string dst\_ip\_addrs[];

bool src\_ip\_required;

bool dst\_ip\_required;

Route src\_routes[];

Route dst\_routes[];

string excluded\_prefixes[];

IpNeighbor ip\_neighbors[];

ExtraPrefixRequest extra\_prefix\_request[];

string extra\_prefixes[];

PolicyRoute policies[];

}



**Route:**

{

string prefix;

string nextHop;

}



**IpNeighbor:**

{

string ip;

string hardware\_address;

}



**ExtraPrefixRequest:**

{

IpFamily addr\_family;

uint32 prefix\_len;

uint32 required\_number;

uint32 requested\_number;

}



**Family:**

enum {

IPV4 = 0;

IPV6 = 1;

}



**IpFamily:**

{

Family family;

}



**PolicyRoute:**

{

string from;

string proto;

string dst\_port;

string src\_port;

Route routes[];

}



**DNSContext**:

{

DNSConfig configs[];

}

**DNSConfig:**

{

string dns\_server\_ips[];

string search\_domains[];

}



**EthernetContext:**

{

string src\_mac;

string dst\_mac;

int32 vlan\_tag;

}



**Path:**

{

uint32 index;

PathSegment path\_segments[];

}



**PathSegment:**

{

string name;

string id;

string token;

Timestamp expires;

map<string, string> metrics;

}



**State:**

enum {

UP = 0;

DOWN = 1;

REFRESH\_REQUESTED = 2;

RESELECT\_REQUESTED = 3;

}



**NetworkServiceEndpoint:**

{

string name;

string network\_service\_names[];

map<string, NetworkServiceLabels> network\_service\_labels;

string url;

Timestamp expiration\_time;

Timestamp initial\_registration\_time;

string path\_ids[];

}



**NetworkServiceLabels:**

{

map<string, string> labels;

}



**Timestamp:**

{

int64 seconds;

int32 nanos;

}



**NetworkService:**

 {

string name;

string payload;

Match matches[];

string path\_ids[];

}



**Match:**

{

map<string, string> source\_selector;

Destination routes[];

bool fallthrough;

Metadata metadata;

}



**Destination:**

{

map<string, string> destination\_selector;

uint32 weight;

}



**Metadata:**

{

map<string, string> labels;

}

