ENVOY - Open source edge and service proxy, designed for cloud-native applications

Filip Kitka, Mateusz Furga
Norbert Morawski, Łukasz Wala
Group 6 (Thursday 13:15), 2024

Kraków, June 6, 2024

Contents

1 Introduction

Envoy [4] proxy is an open-source edge and service proxy designed for cloud-native applications. It is widely used in modern microservices architectures and service mesh implementations. It is often deployed as a sidecar alongside each service instance in a microservices architecture, facilitating communication between services and providing various traffic management functionalities.

This project will attempt to create a case study of the Envoy Proxy. The case study will showcase the features of the Envoy Proxy, such as first class HTTP/2 and gRPC support, the *service mesh* architecture, dynamic service discovery, load balancing, edge proxy support and best in class observability.

2 Theoretical background/technology stack

This chapter will describe some of the core concepts behind the Envoy Proxy, as well as some other technologies used in the case study.

2.1 Envoy Proxy

Envoy proxy is a high-performance open-source project developed for modern cloud-native architectures, particularly microservices. It acts as a sidecar proxy, meaning it runs alongside applications and handles communication between them.

Key features of Envoy proxy:

1. **L7 traffic management:** Envoy can route traffic based on various factors like URLs, headers, and more. It can also handle tasks like load balancing, which distributes traffic evenly across different service instances, and circuit breaking, which prevents overloading services.

2. **Security:** Envoy provides features like TLS termination for encryption and automatic service discovery to ensure applications connect to the correct services securely.

3. **Observability:** Envoy offers extensive monitoring and logging capabilities, making it easier to troubleshoot network issues and gain insights into service communication.

Envoy possible use cases:

1. **Service mesh:** Envoy is a popular choice for building service meshes, which provide a dedicated infrastructure layer for managing communication between services.

2. **API Gateway:** Envoy can act as an API gateway, a single entry point for external clients to access multiple backend services.

2.2 Service mesh

A service mesh is a dedicated infrastructure layer designed to simplify and manage communication between services, particularly in microservice architectures. Provides a separate layer for service communication.

Using service mesh:

1. **Simplified Communication:** The service mesh handles tasks like load balancing, service discovery, and security, offloading these burdens from individual services. This makes development and maintenance easier.
2. **Security:** A service mesh can enforce security policies like encryption and authorization throughout your application, improving overall security posture.
3. **Observability:** Service meshes offer features for monitoring and tracing service communication, providing valuable insights for troubleshooting and performance optimization.

2.3 gRPC

gRPC (gRPC Remote Procedure Calls) is an open-source framework that facilitates high-performance communication between services. gRPC allows to invoke methods on a remote server as if they were local methods on the machine. This simplifies development by enabling writing code without worrying about the underlying network communication details. gRPC is highly performant, language agnostic and strongly typed, in contrast to e.g the typical REST API.

Some use cases for gRPC:

1. **Microservices Communication:** gRPC is a popular choice for building microservice architectures. It enables efficient communication between independent services within an application.
2. **Real-time Communication:** gRPC's streaming capabilities make it suitable for building real-time applications like chat or data feeds.

2.4 Front proxy

In the context of Envoy proxy, a front-proxy [2] refers to a specific deployment configuration where Envoy acts as an intermediary server sitting in front of one or more backend services. It essentially functions as a forward proxy, handling client requests and directing them to the appropriate backend service.

How a front-proxy with Envoy works:

1. **Client Sends Request:** A client (like a web browser) initiates a request by sending it to the Envoy front-proxy.

2. **Envoy Routes Request:** Envoy, based on pre-defined rules, routes the request to the most suitable backend service. This routing can be based on factors like URLs, headers, or load balancing strategies.

3. **Backend Service Processes:** The chosen backend service receives the request from Envoy and processes it.

4. **Response Sent Back:** The backend service sends its response back to Envoy.

5. **Envoy Delivers Response:** Finally, Envoy delivers the response from the backend service to the client.

2.5 Envoy as a Postgres sniffer

Envoy offers a built-in Postgres protocol filter [3]. This filter acts like a sniffer, decoding the communication between Postgres clients and the server without interfering with the actual data flow. It can decode Postgres traffic in its non-SSL form. This means it won't modify or decrypt encrypted communication, ensuring a transparent monitoring approach for your Postgres database.

Using Envoy for Postgres statistics:

1. **Lightweight Monitoring:** Envoy's sniffer approach avoids additional load on the Postgres server, making it a lightweight solution for gathering statistics.

2. **Real-time Insights:** Envoy provides near real-time visibility into Postgres activity, allowing you to monitor performance and identify potential issues quickly.

3. **Improved Observability:** The captured statistics can be integrated with monitoring tools, providing valuable insights into your database's overall health and performance.

2.6 Statistics

Prometheus [7] is an open-source monitoring system designed to scrape metrics from various sources, including Envoy. Envoy exposes metrics related to its own operations and the traffic it handles. Prometheus acts as a time series database, storing the collected metrics over time. This allows you to analyze trends and identify patterns in your monitoring data.

Grafana [5] is an open-source platform specifically designed for data visualization. It integrates seamlessly with Prometheus, allowing you to create dashboards and graphs to visualize the metrics collected by Envoy.

3 Case study concept description

The case study will consist of a typical Envoy Proxy use case—a service mesh with a front proxy, as shown in the Figure 1.

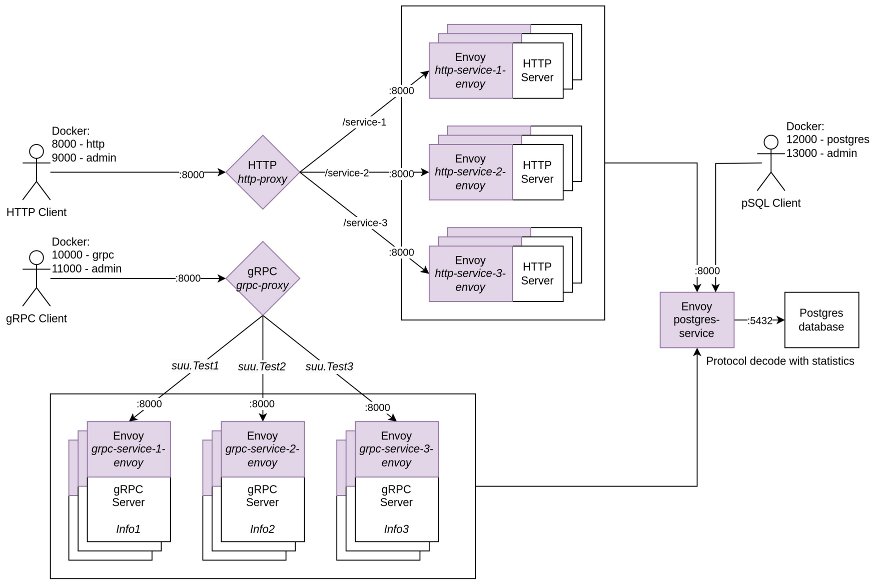


Figure 1: The case study architecture

3.1 Front Proxy

One instance of Envoy will serve as an edge reverse proxy and, therefore, be the entry point for incoming HTTP requests. It will:

1. route the HTTP requests to appropriate services,
2. load balance the requests within groups of the same service.

Another Envoy instance will manage gRPC connections. It will perform analogous task as the proxy defined above.

3.2 HTTP/REST Services

The edge proxy will route to groups of services based on HTTP request paths. Within each group, services will be duplicated (to enable load balancing) and respond to HTTP requests, sometimes also communicating with other services in the mesh, such as gRPC services and the database. All services will include a sidecar instance of Envoy.

3.3 gRPC Services

Just as with HTTP services, the mesh will contain a group of gRPC services. These services will also be available via (separate) edge proxy. The gRPC services group may also include an internal load balancer. All services will include a sidecar instance of Envoy.

3.4 Database

The mesh will include a single instance of the Postgres database that might be queried by both the HTTP and gRPC services. The database will also include a sidecar instance of Envoy that can decode the Postgres protocol and collect statistics on the database queries.

3.5 Additions

The simple case study can be expanded with:

1. an additional edge proxy to create the "Double proxy" setup that allows for client/TLS connection termination closer (geographically) to the user, which offers benefits such as a shorter round-trip time for the TLS handshake.
2. rate limiting on the edge proxy (which, in real use cases, can prevent DDoS attacks),
3. proxying WebSocket connections by the edge proxy.

The service mesh will come with built-in support for functionalities like service discovery, health checks, and dynamic configuration (using, for example, the go-control-plane). The case study will also leverage Envoy's observability features by setting up Grafana visualization, which will utilize Prometheus metrics.

4 Solution architecture

Below, we describe vital components and supported protocols.

4.1 HTTP services

To implement HTTP services, Python will be used along with the FastAPI framework which provides a user-friendly, modern, and fast interface for creating web APIs. The created services will implement several simple endpoints, which will return basic information about the service (such as: service name, service IP address, and request URL) in JSON format.

4.2 gRPC services

To implement gRPC services, Python language will be used along with the grpc_tools tool, which will be used to generate server-side code based on the service definition in a proto file. The gRPC services will implement a few simple methods that will return basic information about the service, similar to the HTTP services.

4.3 Database

Based on Envoy's Docker image, a container is built with additional Postgres install. Envoy proxies access to DB instance, while also collecting statistics. A simple database initialization script is provided which establishes some test data inside the tables.

4.4 Statistics

Grafana and Prometheus will be configured to collect and visualize statistics from Envoy and Postgres. Custom dashboards in Grafana will be created to present a view of the system's load and activity levels.

4.5 Build

For all the above-mentioned services, appropriate Docker images and Docker Compose configuration files will be created. Utilizing these tools will greatly facilitate application build, subsequent deployment, and the reproduction of conducted tests, both in the local environment and in the cloud.

5 Environment configuration description

The configuration of the Envoy proxies used throughout the system will be described in this section (along with some configuration code snippets), as well as the configuration of other elements, like the dummy HTTP and gRPC services, or the observability tools.

5.1 HTTP Front Proxy

The *Front Proxy* will be deployed on the edge of the service mesh on a separate machine as a Docker container. The configuration consists of `static_resources` section describing the address and port used to listen for the HTTP traffic:

```
listeners:
- address:
    socket_address:
      address: 0.0.0.0
      port_value: 8000
```

as well as the `filter_chains` section describing HTTP routing rules[1]:

```
route_config:
  name: local_route
  virtual_hosts:
  - name: backend
    domains:
    - "*"
    routes:
    - match:
        prefix: "/service-1"
      route:
        cluster: http-service-1-envoy
```

The clusters are described in the `clusters` section, which describes the addresses and ports used by the HTTP service sidecar proxies, as well as load balancing policies. Here's is the examples cluster section:

```
- name: http-service-1-envoy
  type: STRICT_DNS
  lb_policy: ROUND_ROBIN
  load_assignment:
    cluster_name: http-service-1-envoy
    endpoints:
    - lb_endpoints:
      - endpoint:
          address:
            socket_address:
              address: http-service-1-envoy
              port_value: 8000
```

5.2 HTTP Service Sidecar Proxy

This Envoy instance will be deployed alongside all HTTP services in the mesh. It provides access to the app hosted locally on the machine. To open necessary port, `socket_address` instance is created:

```
- address:
  socket_address:
    address: 0.0.0.0
    port_value: 8000
```

Routing is achieved by matching all prefixes and forwarding them to the cluster referencing `localhost:20000` where Web server resides:

[1]Only a single route included in the snippet for the sake of conciseness.

```
route_config:
          name: local_route
          virtual_hosts:
          - name: backend
            routes:
            - match:
                prefix: ""
              route:
                cluster: localhost-app

clusters:
  - name: localhost-app
    type: STRICT_DNS
    lb_policy: ROUND_ROBIN
    load_assignment:
      cluster_name: localhost-app
      endpoints:
      - lb_endpoints:
        - endpoint:
            address:
              socket_address:
                address: localhost
                port_value: 20000
```

Also, Envoy sidecar enables the app to access Postgres database service at `localhost:5432`:

```
- address:
  socket_address:
    address: 127.0.0.1
    port_value: 5432
filter_chains:
  - filters:
    - name: envoy.tcp_proxy
      typed_config:
        "@type": type.googleapis.com/
                 envoy.extensions.filters.network.tcp_proxy.v3.TcpProxy
        stat_prefix: tcp
        cluster: postgres-db
```

Python program will not be required to know exact DB server's location. A simple local socket will suffice.

5.3 gRPC proxy and sidecars

gRPC proxy runs almost exactly like it's HTTP counterpart. The routes are configured as follows:

```
routes:
- match:
    prefix: "/suu.Test1"
    grpc: {}
  route:
    cluster: grpc-service-1-envoy
- match:
    prefix: "/suu.Test2"
    grpc: {}
  route:
    cluster: grpc-service-2-envoy
- match:
    prefix: "/suu.Test3"
    grpc: {}
  route:
    cluster: grpc-service-3-envoy
```

The configuration of sidecars is analogous to HTTP sidecars.

5.4 Postgres database service

The database runs locally on port `5432` and the proxy exposes is to the mesh via:

```
- address:
  socket_address:
    address: 0.0.0.0
    port_value: 8000
```

Thus, DB service is accessible on port `8000` externally.
Also, the container features simple `init.sql` script:

```
CREATE USER envoy WITH PASSWORD 'envoy';
CREATE DATABASE envoy;
GRANT ALL PRIVILEGES ON DATABASE envoy TO envoy;

CREATE TABLE cars (
  brand VARCHAR(255),
  model VARCHAR(255),
  year INT
);

GRANT ALL PRIVILEGES ON TABLE cars TO envoy;

INSERT INTO cars VALUES ('Audi','A6',1998);
INSERT INTO cars VALUES ('Peugeot','206',2006);
INSERT INTO cars VALUES ('Renault','Captur',2018);
```

This data is accessed via HTTP and gRPC services and displayed in responses for the user to verify.

6 Installation method

As already mentioned, Docker/Docker Compose is being used. It can quickly produce testing environment. Also, simple scaling of requested backend services is supported.

In the root directory of the repository, a main `docker-compose.yaml` file is located. It includes all other Compose files:

```
include:
  - http-proxy/docker-compose.yaml
  - http-services/docker-compose.yaml
  - grpc-proxy/docker-compose.yaml
  - grpc-services/docker-compose.yaml
  - postgres-service/docker-compose.yaml
  - grafana/docker-compose.yaml
```

Below, we describe those files and their respective custom `Dockerfile`, if present.

6.1 Envoy proxy instance files

This files, located in `http-proxy` and `grpc-proxy`, are quite straightforward. Here is a HTTP proxy configuration file (gRPC one is identical except for exposed ports):

```
services:
  http-proxy:
    image: envoyproxy/envoy:v1.30.1
    volumes:
      - ./http-proxy-envoy.yaml:/envoy.yaml
    command:
      -c /envoy.yaml
    ports:
      - 8000:8000 # http proxy
      - 9000:9000 # admin interface
```

We use official Envoy image `envoyproxy/envoy` version `v1.30.1`. A config file is mounted at `/envoy.yaml` and the instance is informed to use it with `-c /envoy.yaml` command line option. A set of ports is exposed:

- `8000` – HTTP proxy main interface
- `9000` – HTTP proxy admin interface

- `10000` – gRPC proxy main interface
- `11000` – gRPC proxy admin interface

6.2 Envoy service instance files

Docker Compose files contained in `http-services` and `grpc-services` contain a set of service definitions (only one showed for simplicity):

```
services:
  http-service-1-envoy:
    build: .
    volumes:
      - ./http-sidecar-envoy.yaml:/envoy.yaml
    environment:
      SERVICE_NAME: service-1
    depends_on:
      postgres-service:
        condition: service_healthy
```

It also bind mounts a config inside the container. Additionally, it sets a service name reported by the API. Also, it is made sure, database service starts and is healthy before any application servers come up.

Here, a custom Docker image is built:

```
FROM envoyproxy/envoy:v1.30.1

# install python
RUN apt update && \
 apt install -y --no-install-recommends python3 python3-pip

# install service.py and requirements
WORKDIR /code

COPY ./requirements.txt /code/requirements.txt
RUN pip install --no-cache-dir --upgrade -r /code/requirements.txt

COPY ./service.py /code/

# start envoy in background (&)
# replace shell with uvicorn (exec, forward SIGTERM to uvicorn)
# listen both on v4 and v6
CMD ["/bin/bash", "-c",
     "envoy -c /envoy.yaml &
      exec uvicorn service:app
        --host 127.0.0.1
        --host ::1
        --port 20000"]
```

It is based on Envoy image, which itself is a Debian-like distribution (hence `apt`). After Python and dependency installation, a shell starts a background task of Envoy and replaces itself with a Python HTTP server. It listens both on IPv4 and IPv6 loopback addresses, as some `localhost` host names resolve to IPv6. gRPC `Dockerfile` is almost the same, except for protocol compilation:

```
RUN python3 -m grpc_tools.protoc -I.
   --python_out=.
   --grpc_python_out=.
   info.proto
```

6.3 Postgres database

Below is a Docker Compose file responsible for building and running Docker instance of Postgres and its sidecar Envoy instance:

```
services:
  postgres-service:
    build: .
    volumes:
      - ./postgres-sidecar-envoy.yaml:/envoy.yaml
      - ./pg_hba.conf:/var/lib/postgresql/data/pg_hba.conf
      - ./postgresql.conf:/var/lib/postgresql/data/postgresql.conf
    ports:
      - 12000:8000 # db access with psql via envoy
      - 13000:9000 # admin
    healthcheck:
        test: ["CMD", "psql"]
        interval: 1s
        timeout: 1s
        retries: 60
```

Additionally to similar `volumes` and `ports` sections, a `healthcheck` is defined. It provides a way to check whether the Postgres daemon is ready to accept queries from the application servers.

As with HTTP and gRPC, it requires a custom Docker build file:

```
# contrib required for postgres plugin
FROM envoyproxy/envoy:contrib-v1.30.1

# install postgres
RUN apt update && \
 apt install -y --no-install-recommends postgresql

ENV PATH="${PATH}:/usr/lib/postgresql/14/bin"

ENV PGDATA=/var/lib/postgresql/data
```

```
USER postgres
RUN initdb

COPY ./init.sql /tmp/init.sql
RUN postgres & sleep 1 && psql < /tmp/init.sql

# start envoy in background (&)
# replace shell with postgres (exec)
CMD ["/bin/bash", "-c", "envoy -c /envoy.yaml & exec postgres"]
```

Now, we're using community-maintained `contrib-v1.30.1` version of the Envoy image. It is required, because only this release features a work-in-progress Postgres filter.

The image takes care of DB provision and running user-provided init script. Finally, a command is defined to run Envoy as background task and also run postgres as the main process.

6.4 Grafana and Prometheus

The following configurations set up Grafana and Prometheus services for monitoring and visualization. Below is the Docker Compose configuration for Grafana:

```
services:
  grafana:
    image: grafana/grafana-enterprise
    restart: unless-stopped
    volumes:
      - grafana_data:/var/lib/grafana
      - ./provisioning:/etc/grafana/provisioning
      - ./dashboards:/etc/grafana/dashboards
    ports:
     - 3000:3000
```

- `grafana_data:/var/lib/grafana` – Persistent storage for Grafana data
- `./provisioning:/etc/grafana/provisioning` – Configuration for provisioning
- `./dashboards:/etc/grafana/dashboards` – Pre-configured dashboards

Port `3000` is exposed for accessing the Grafana web interface.

Next, here is the next services section of Docker Compose configuration for Prometheus:

```
  prometheus:
    image: prom/prometheus
    restart: unless-stopped
    volumes:
```

```
    - ./prometheus.yaml:/etc/prometheus/prometheus.yml
    - prometheus_data:/prometheus
  command:
    - --config.file=/etc/prometheus/prometheus.yml
    - --storage.tsdb.path=/prometheus
    - --web.console.libraries=/etc/prometheus/console_libraries
    - --web.console.templates=/etc/prometheus/consoles
    - --web.enable-lifecycle
  ports:
    - 9090:9090
```

- `./prometheus.yaml:/etc/prometheus/prometheus.yml` – Main configuration file
- `prometheus_data:/prometheus` – Persistent storage for time series data

Command line options configure Prometheus to use the specified configuration file, storage path, and enable the web lifecycle. Port `9090` is exposed for accessing the Prometheus web interface.

7 How to reproduce - step by step

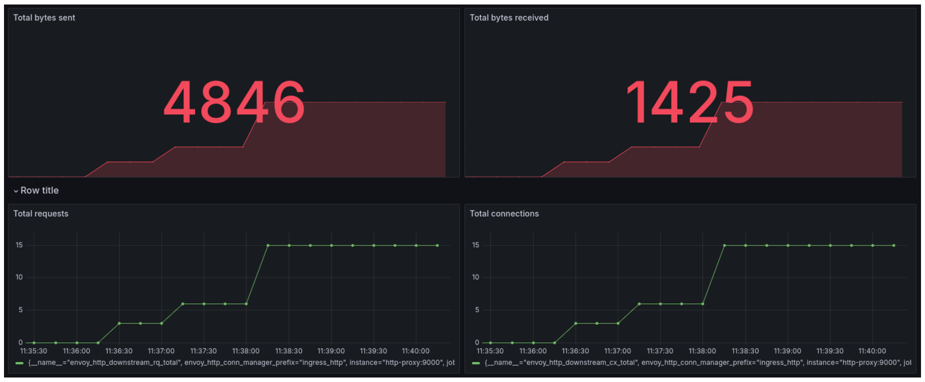
Here we present a list of steps to run our demo:

1. Install Docker, follow [1],
2. Install `git` and clone the repository: `git clone https://github.com/mfurga/suu-envoy.git`
3. Enter the folder: `cd suu-envoy`,
4. Run Docker Compose: `docker compose up -d`,
5. (Testing) Install `grpcurl` command-line gRPC client (on Debian-like download correct deb file from [6] and install via `dpkg -i <deb file>`),
6. (Testing) Install `psql` Postgres client (on Debian-like install `postgresql-client`),
7. (Testing) Run provided test script: `./test.sh`,
8. (Testing) You can scale backend HTTP and gRPC services for example by issuing `docker compose scale http-service-1-envoy=3`.
9. Go to 127.0.0.1:3000 to see the Grafana dashboard (7)

Test script should produce something like this:

```
HTTP service-1:
{"service":"service-1",
"url":"http://127.0.0.1:8000/service-1/dbquery","ip":"172.21.0.13",
"db":[["Audi","A6",1998],["Peugeot","206",2006],["Renault","Captur",2018]]}
```

```
HTTP service-2:
{"service":"service-2",
"url":"http://127.0.0.1:8000/service-2/dbquery","ip":"172.21.0.11",
"db":[["Audi","A6",1998],["Peugeot","206",2006],["Renault","Captur",2018]]}

HTTP service-3:
{"service":"service-3",
"url":"http://127.0.0.1:8000/service-3/dbquery","ip":"172.21.0.12",
"db":[["Audi","A6",1998],["Peugeot","206",2006],["Renault","Captur",2018]]}


gRPC service-1:
{
  "name": "Test1",
  "ip": "172.21.0.9"
}

gRPC service-2:
{
  "name": "Test2",
  "ip": "172.21.0.3"
}

gRPC service-3:
{
  "name": "Test3",
  "ip": "172.21.0.2"
}

Postgres service:
  brand  | model  | year
---------+--------+------
 Audi    | A6     | 1998
 Peugeot | 206    | 2006
 Renault | Captur | 2018
(3 rows)
```



```
{__name__="envoy_http_downstream_rq_total", envoy_http_conn_manager_prefix="ingress_http", instance="http-proxy:9000", jol
{__name__="envoy_http_downstream_cx_total", envoy_http_conn_manager_prefix="ingress_http", instance="http-proxy:9000", jol
```

Figure 2: Grafana dashboard

8 Summary

Project demonstrates the capabilities of the Envoy proxy, as a sidecar proxy. By setting up a service mesh, dynamic routing, load balancing and observability, the case study provides an overview of Envoy's features and applications.

The use of Docker and Docker Compose simplifies the deployment process, making it easy to reproduce the setup for further experiments or production environments.

Project confirms how Envoy is effective in managing service-to-service communication and improving observability in microservice deployments.

References

[1] *Docker installation tutorial*. URL: `https://docs.docker.com/engine/install` (visited on 06/05/2024).

[2] *Envoy front proxy example*. URL: `https://www.envoyproxy.io/docs/envoy/latest/start/sandboxes/front_proxy` (visited on 06/05/2024).

[3] *Envoy Postgres filter*. URL: `https://www.envoyproxy.io/docs/envoy/latest/configuration/listeners/network_filters/postgres_proxy_filter` (visited on 06/05/2024).

[4] *Envoy's home page*. URL: `https://www.envoyproxy.io/` (visited on 06/05/2024).

[5] *Grafana*. URL: `https://grafana.com/docs/grafana/latest/` (visited on 06/05/2024).

[6] *grpcurl release files*. URL: `https://github.com/fullstorydev/grpcurl/releases/` (visited on 06/05/2024).

[7] *Prometheus as datasource in Grafana*. URL: `https://prometheus.io/docs/visualization/grafana/` (visited on 06/05/2024).