AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

Wydział Informatyki

INSTYTUT INFORMATYKI

Środowiska Udostępniania Usług

Network Service Mesh

Kacper Iwicki

Mikołaj Maślak

Anna Nowacka

Kraków, 2025

Spis treści

Rozdział 1

Wprowadzenie

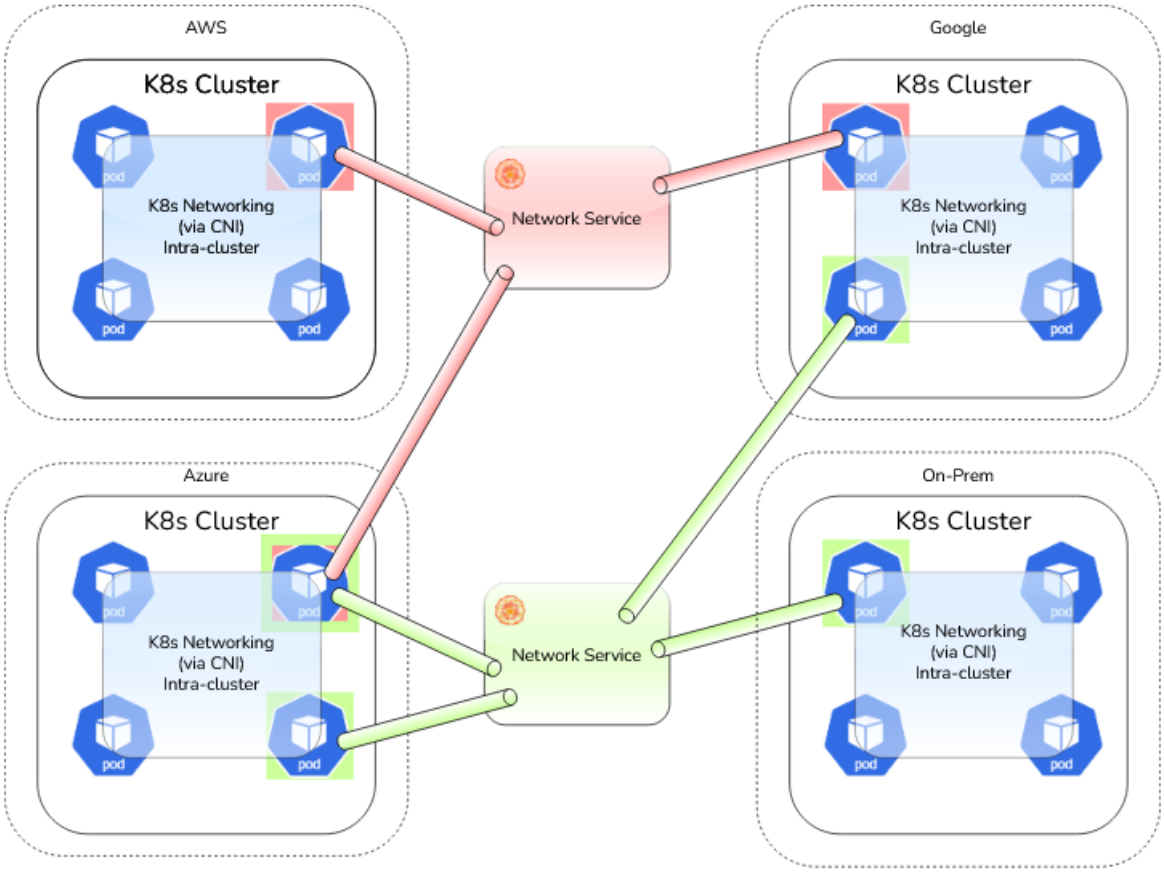
W nowoczesnych środowiskach chmurowych rośnie zapotrzebowanie na elastyczne, skalowalne i bezpieczne rozwiązania sieciowe, co doprowadziło do rozwoju technologii typu Service Mesh. Niniejszy projekt koncentruje się na wykorzystaniu Network Service Mesh [1] (NSM) – unikatowego podejścia do zarządzania złożonymi połączeniami sieciowymi w środowisku Kubernetes. Celem projektu jest zbadanie, w jaki sposób NSM umożliwia tworzenie niestandardowych topologii sieciowych, zwiększa bezpieczeństwo oraz poprawia obserwowalność ruchu sieciowego w aplikacji rozproszonej.

Rozdział 2

Podstawy teoretyczne i stos technologiczny

2.1. Podstawy teoretyczne

Network Service Mesh to narzędzie do zarządzania łącznością sieciową w środowiskach opartych na Kubernetes. W przeciwieństwie do tradycyjnych Service Meshów, które działają na warstwie aplikacji (warstwa 7 modelu OSI), NSM operuje na warstwie niższej - 3, co pozwala na transport pakietów IP oraz wspieranie legacy systemów używających nietypowych protokołów do komunikacji. NSM może służyć do łączenia aplikacji działających on-premise z tymi w chmurze, a także do współpracy aplikacji działających w różnych środowiskach chmurowych.



Rysunek 2.1: Przykład architektury używającej Network Service Mesh (NSM). Źródło: https://networkservicemesh.io/docs/concepts/enterprise_users/

2.2. Użyte Technologie

W projekcie zostały użyte następujące technologie:

- **Node.js** – środowisko uruchomieniowe JavaScript do tworzenia aplikacji serwerowych.
- **Express.js** – minimalistyczny framework dla Node.js, upraszczający tworzenie API i serwerów.
- **OpenTelemetry SDK** [2] – zestaw narzędzi do zbierania metryk, śledzenia i logów aplikacji.
- **Prometheus** – system monitoringu i zbierania metryk dla aplikacji i infrastruktury.
- **Tempo** – system zbierający traces aplikacji.
- **OpenTelemetry Collector** – komponent zbierający i przetwarzający dane telemetryczne z różnych źródeł.
- **Grafana** – platforma do wizualizacji danych telemetrycznych i monitoringu.
- **Kubernetes** – system orkiestracji kontenerów do automatyzacji wdrażania, skalowania i zarządzania aplikacjami.

Rozdział 3

Opis studium przypadku

Stworzona aplikacja to system symulujący proces zamawiania i dostarczania jedzenia w środowisku opartym na trzech mikroserwisach:

- `Restaurant Service`
- `Orders Service`
- `Delivery Service`

Komponenty działają w różnych przestrzeniach nazw w klastrze Kubernetes. Do komunikacji między mikroserwisami zastosowano **Network Service Mesh (NSM)**, który pozwala na tworzenie dynamicznych, prywatnych ścieżek sieciowych między usługami bez potrzeby użycia globalnych adresów IP czy tradycyjnego routingu w ramach klastra. Komunikacja odbywa się z wykorzystaniem deklaratywnie zdefiniowanych usług sieciowych, zapewniając przy tym kontrolę dostępu oraz lepsze możliwości obserwowalności.

Scenariusz użycia

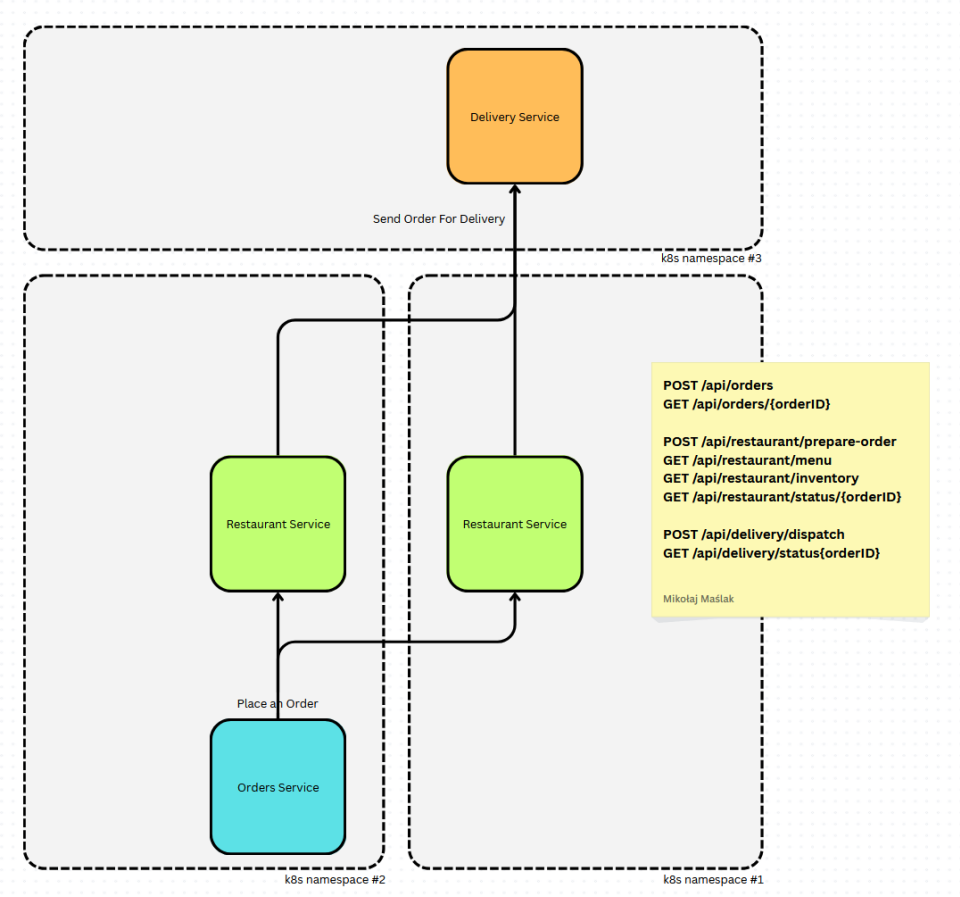
Przypadek użycia aplikacji obejmuje pełny cykl realizacji zamówienia:

1. **Złożenie zamówienia**: Użytkownik inicjuje zamówienie przez usługę `orders-service`, która zbiera dane o zamówieniu.
2. **Weryfikacja i przygotowanie**: `orders-service` kontaktuje się z `restaurant-service` w celu potwierdzenia dostępności produktów i przygotowania zamówienia.
3. **Dostarczenie zamówienia**: Gdy zamówienie jest gotowe, `delivery-service` przejmuje je i realizuje dostawę do końcowego odbiorcy.

Na każdym etapie komunikacja między usługami przebiega przez łącza zestawione dynamicznie przez NSM, co pozwala na pełną kontrolę i obserwowalność przepływu żądań. Dzięki temu możliwa jest analiza wydajności systemu, wykrywanie opóźnień sieciowych oraz testowanie niezawodności połączeń między usługami bez ich fizycznej ekspozycji na zewnątrz klastra.

Rozdział 4

Architektura rozwiązania



Rysunek 4.1: Schemat architektury mikroserwisów

Diagram 4.1 przedstawia ogólną strukturę komponentów oraz kierunki komunikacji między mikroserwisami.

Opis endpointów

Każdy mikroserwis udostępnia zestaw endpointów REST API. Poniżej opisano główne funkcjonalności każdego z serwisów z uwzględnieniem ich interfejsów:

Orders Service

- `POST /api/orders` – tworzy nowe zamówienie.
- `GET /api/orders/{orderID}` – zwraca szczegóły zamówienia o danym identyfikatorze.

Restaurant Service

- `POST /api/restaurant/prepare-order` – inicjuje proces przygotowania zamówienia.
- `GET /api/restaurant/menu` – zwraca aktualne menu.
- `GET /api/restaurant/inventory` – zwraca stan magazynowy produktów.
- `GET /api/restaurant/status/{orderID}` – zwraca status realizacji konkretnego zamówienia.

Delivery Service

- `POST /api/delivery/dispatch` – zleca rozpoczęcie dostawy.
- `GET /api/delivery/status/{orderID}` – zwraca status dostawy zamówienia.

Rozdział 5

Konfiguracja środowiska

W ramach projektu skonfigurowano lokalne środowisko deweloperskie oraz klaster Kubernetes. Poniżej przedstawiono szczegółowe kroki konfiguracji na różnych etap projektu.

5.1. Wymagania systemowe

Konieczne było zainstalowanie poniższych komponentów

- **Node.js** w wersji 16 lub nowszej
- **Docker** do konteneryzacji aplikacji
- **kubectl** do zarządzania klastrem Kubernetes
- **Minikube** do uruchomienia lokalnego klastra Kubernetes

5.2. Konfiguracja Node.js i zależności

W celu przetestowania aplikacji można ją uruchomić lokalnie z Node.js. Pierwszym krokiem jest instalacja zależności aplikacji:

```
npm install
```

Listing 5.1: Instalacja zależności Node.js

Po zainstalowaniu zależności można uruchomić wszystkie mikroserwisy w trybie deweloperskim:

```
npm run start
```

Listing 5.2: Uruchomienie serwisów lokalnie

Ta komenda uruchamia równocześnie wszystkie trzy mikroserwisy:

- Restaurant Service na porcie 3001
- Orders Service na porcie 3002
- Delivery Service na porcie 3003

Nie jest to jednak prawidłowe podejście do uruchamiania skalowalnych aplikacji, a jedynie w celach deweloperskich.

5.3. Konfiguracja Docker

W ramach realizacji projektu wykorzystano Docker do konteneryzacji mikroserwisów. Każdy serwis ma wspólny plik Dockerfile z parametryzacją przez argument `SERVICE`:

```
docker build -f services/Dockerfile -t restaurant-service:latest
    --build-arg SERVICE=restaurant .
docker build -f services/Dockerfile -t orders-service:latest --build-arg
    SERVICE=orders .
docker build -f services/Dockerfile -t delivery-service:latest --build-arg
    SERVICE=delivery .
```

Listing 5.3: Budowanie obrazów Docker

Można sprawdzić działanie poprzez uruchomienie kontenerów w środowisku Docker:

```
docker run -p 3001:3001 --env-file .env.docker-local
    restaurant-service:latest
docker run -p 3002:3002 --env-file .env.docker-local orders-service:latest
docker run -p 3003:3003 --env-file .env.docker-local
    delivery-service:latest
```

Listing 5.4: Uruchomienie kontenerów

W ten sposób można ominąć lokalne uruchamianie oprogramowania, tworząc zamiast tego dedykowane kontenery.

5.4. Konfiguracja systemu monitoringu

Projekt wykorzystuje stos OTEL-LGTM (OpenTelemetry - Loki, Grafana, Tempo, Prometheus) do monitoringu i obserwowalności. Uruchomienie systemu monitoringu w skonteneryzowanym środowisku jest bardzo proste:

```
docker run -p 3000:3000 -p 4317:4317 -p 4318:4318 grafana/otel-lgtm
```

Listing 5.5: Uruchomienie stosu monitoringu

Stos ten obejmuje:

- **Grafana** (port 3000) – dashboard do wizualizacji
- **Tempo** – backend do przechowywania traces
- **Prometheus** – system metryk
- **Loki** – system logów
- **OpenTelemetry Collector** (porty 4317, 4318) – kolektor telemetrii, wykorzystywany przez serwisy

Rozdział 6

Sposób instalacji

6.1. Instalacja Minikube

Minikube umożliwia uruchomienie lokalnego klastra Kubernetes. Szczegółowy proces instalacji znajduje się w oficjalnej dokumentacji:
https://minikube.sigs.k8s.io/docs/handbook/controls/
Po instalacji należy uruchomić klaster:

```
minikube start
```

Listing 6.1: Uruchomienie Minikube

6.2. Przygotowanie obrazów kontenerów

Istnieją dwa sposoby przygotowania obrazów kontenerów w środowisku Minikube:

6.2.1. Opcja 1: Ładowanie wcześniej zbudowanych obrazów

Jeśli obrazy zostały wcześniej zbudowane lokalnie:

```
minikube image load restaurant-service
minikube image load orders-service
minikube image load delivery-service
```

Listing 6.2: Ładowanie obrazów do Minikube

6.2.2. Opcja 2: Budowanie wewnątrz Minikube

Alternatywnie można użyć dostarczonego skryptu do budowania obrazów bezpośrednio w środowisku Minikube:

```
./build-containers.sh
```

Listing 6.3: Budowanie obrazów w Minikube

Ten skrypt automatycznie konfiguruje środowisko Docker Minikube i buduje wszystkie wymagane obrazy.

6.3. Instalacja Network Service Mesh

Network Service Mesh wymaga instalacji dedykowanych komponentów w klastrze Kubernetes. Proces instalacji NSM obejmuje:

1. Instalację kontrolera NSM
2. Konfigurację węzłów roboczych (forwarder planes)
3. Definicję usług sieciowych (Network Services)
4. Konfigurację klientów i punktów końcowych

```
kubectl apply -f k8s/nsm-install.yaml
```

Listing 6.4: Instalacja komponentów systemowych NSM

Rozdział 7

Odtworzenie rozwiązania - krok po kroku

7.1. Infrastructure as Code (IaC) - Podejście

Projekt wykorzystuje podejście Infrastructure as Code (IaC) do zarządzania całą infrastrukturą aplikacji. IaC polega na definiowaniu infrastruktury w plikach tekstowych zamiast manualnych procesów konfiguracyjnych.

Implementacja IaC w projekcie

Cała infrastruktura systemu jest zdefiniowana deklaratywnie w plikach YAML Kubernetes znajdujących się w katalogu `k8s/`:

- **Definicje deploymentów** – `restaurant.yaml`, `orders.yaml`, `delivery.yaml`
- **Konfiguracja Network Service Mesh** – `nsm-install.yaml`, `nsm-services.yaml`, `nsm-clients.yaml`
- **Zarządzanie namespace'ami** – `namespaces.yaml`
- **Konfiguracja aplikacji** – `shared-config.yaml` (ConfigMaps)
- **Infrastruktura monitoringu** – `otel-lgtm.yaml`

Automatyzacja wdrożeń

Projekt zawiera skrypty automatyzujące proces wdrożenia:

```
./build-containers.sh
```

Listing 7.1: Automatyczne budowanie kontenerów

```
kubectl apply -f k8s/
```

Listing 7.2: Wdrożenie całej infrastruktury

7.2. Przygotowanie środowiska

7.2.1. Krok 1: Klonowanie repozytorium

```
git clone git@github.com:M0rgho/suu-ns-mesh.git
cd suu-ns-mesh
```

Listing 7.3: Klonowanie projektu

7.2.2. Krok 2: Instalacja zależności

```
npm install
```

Listing 7.4: Instalacja zależności

7.2.3. Krok 3: Testowanie lokalnie

Przed wdrożeniem w Kubernetes warto przetestować aplikację lokalnie:

```
npm run start
```

Listing 7.5: Test lokalny

Sprawdzenie działania przez wysłanie przykładowego żądania:

```
curl -X POST \
  -H "Content-Type: application/json" \
  -d '{"items": [{"name": "Pizza", "quantity": 1}]}' \
  http://localhost:3002/api/orders
```

Listing 7.6: Test żądania

7.3. Wdrożenie w środowisku Kubernetes

7.3.1. Krok 4: Uruchomienie klastra

```
minikube start
```

Listing 7.7: Start klastra Minikube

7.3.2. Krok 5: Instalacja Network Service Mesh

```
kubectl apply -f k8s/nsm-install.yaml

# Oczekiwanie na gotowosc komponentow
kubectl wait --for=condition=ready pod -l app=nsm-registry -n nsm-system
    --timeout=300s
```

```
kubectl wait --for=condition=ready pod -l app=nsm-node -n nsm-system
    --timeout=300s
```

Listing 7.8: Instalacja komponentów NSM

7.3.3. Krok 6: Utworzenie namespace'ów

```
kubectl apply -f k8s/namespaces.yaml
```

Listing 7.9: Tworzenie przestrzeni nazw

7.3.4. Krok 7: Wdrożenie współdzielonej konfiguracji

```
kubectl apply -f k8s/shared-config.yaml -n restaurant-ns
kubectl apply -f k8s/shared-config.yaml -n orders-ns
kubectl apply -f k8s/shared-config.yaml -n delivery-ns
kubectl apply -f k8s/shared-config.yaml -n otel-ns
```

Listing 7.10: Konfiguracja wszystkich namespace'ów

7.3.5. Krok 8: Konfiguracja Network Service Mesh

```
kubectl apply -f k8s/nsm-services.yaml
kubectl apply -f k8s/nsm-clients.yaml
```

Listing 7.11: Definicje usług i klientów NSM

7.3.6. Krok 9: Przygotowanie obrazów

```
# Budowanie obrazow
docker build -f services/Dockerfile -t restaurant-service:latest
    --build-arg SERVICE=restaurant .
docker build -f services/Dockerfile -t orders-service:latest --build-arg
    SERVICE=orders .
docker build -f services/Dockerfile -t delivery-service:latest --build-arg
    SERVICE=delivery .

# Ladowanie do Minikube
minikube image load restaurant-service:latest
minikube image load orders-service:latest
minikube image load delivery-service:latest
```

Listing 7.12: Budowanie i ładowanie obrazów

Alternatywnie można użyć dostarczonego skryptu:

```
./build-containers.sh
```

Listing 7.13: Automatyczne budowanie

7.3.7. Krok 10: Wdrożenie aplikacji

```
kubectl apply -f k8s/restaurant.yaml
kubectl apply -f k8s/orders.yaml
kubectl apply -f k8s/delivery.yaml
kubectl apply -f k8s/otel-lgtm.yaml
```

Listing 7.14: Wdrożenie w Kubernetes

7.3.8. Krok 11: Weryfikacja wdrożenia

Sprawdzenie statusu podów:

```
kubectl get pods --all-namespaces
```

Listing 7.15: Sprawdzenie statusu

Weryfikacja Network Service Mesh:

```
kubectl get networkserviceendpoints -A
kubectl get networkserviceclients -A
kubectl logs -n nsm-system -l app=nsm-registry
```

Listing 7.16: Sprawdzenie NSM

7.4. Konfiguracja dostępu

7.4.1. Dostęp do Grafana

```
kubectl port-forward -n otel-ns svc/otel-lgtm-external 3000:3000
```

Listing 7.17: Port-forward dla Grafana

Po wykonaniu tej komendy Grafana będzie dostępna pod adresem http://localhost:3000.

7.4.2. Dostęp do Orders Service

```
kubectl port-forward service/orders-service 3001:3000
```

Listing 7.18: Port-forward dla Orders Service

7.4.3. Alternatywny dostęp przez Minikube tunnel

W przypadku problemów z port-forward można użyć tunelu Minikube:

```
minikube tunnel
```

Listing 7.19: Minikube tunnel

Więcej informacji: https://minikube.sigs.k8s.io/docs/commands/tunnel/

7.5. Testowanie rozwiązania

7.5.1. Test funkcjonalności

Wysłanie przykładowego zamówienia:

```
curl -X POST \
  -H "Content-Type: application/json" \
  -d '{"items": [{"name": "Pizza", "quantity": 1}]}' \
  http://localhost:3001/api/orders
```

Listing 7.20: Test zamówienia

7.5.2. Monitorowanie

Po wysłaniu żądania można obserwować:

- Traces w interfejsie Grafana
- Metryki wydajności
- Logi komunikacji między serwisami
- Przepływ żądań przez NSM

Rozdział 8

Deployment

8.1. Strategia wdrażania

Projekt wykorzystuje strategię wdrażania opartą na konteneryzacji z użyciem Kubernetes jako platformy orkiestracyjnej. Architektura wdrożenia obejmuje:

- **Separację namespace'ów** – każdy mikroserwis działa w dedykowanej przestrzeni nazw
- **Network Service Mesh** – do zarządzania komunikacją między serwisami
- **Centralizowany monitoring** – jeden stos OTEL-LGTM dla całej aplikacji
- **Konfigurację deklaratywną** – wszystkie komponenty zdefiniowane w plikach YAML

8.2. Struktura plików konfiguracyjnych

Katalog `k8s/` zawiera kompletną konfigurację wdrożenia:

- `delivery.yaml`, `orders.yaml`, `restaurant.yaml` – definicje deploymentów oraz serwisów Kubernetes dla mikroserwisów
- `otel-lgtm.yaml` – konfiguracja stosu monitoringu
- `shared-config.yaml` – wspólna konfiguracja zmiennych środowiskowych

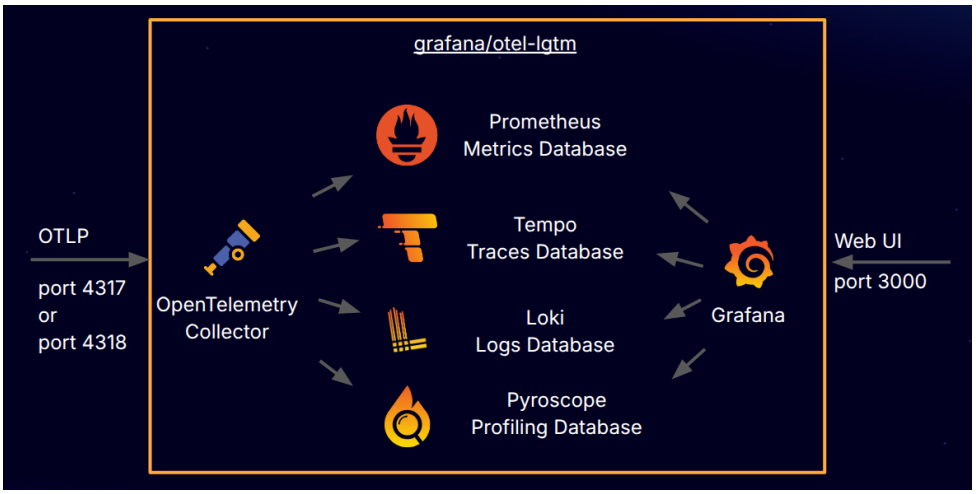
8.3. Zarządzanie konfiguracją

Projekt wykorzystuje zmienne środowiskowe do konfiguracji:

- `.env.docker-local` – konfiguracja dla środowiska Docker
- ConfigMaps Kubernetes
- Argumenty buildowe Docker – dla parametryzacji obrazów

8.4. Monitoring i obserwowalność

Wdrożenie obejmuje kompletny stos monitoringu:



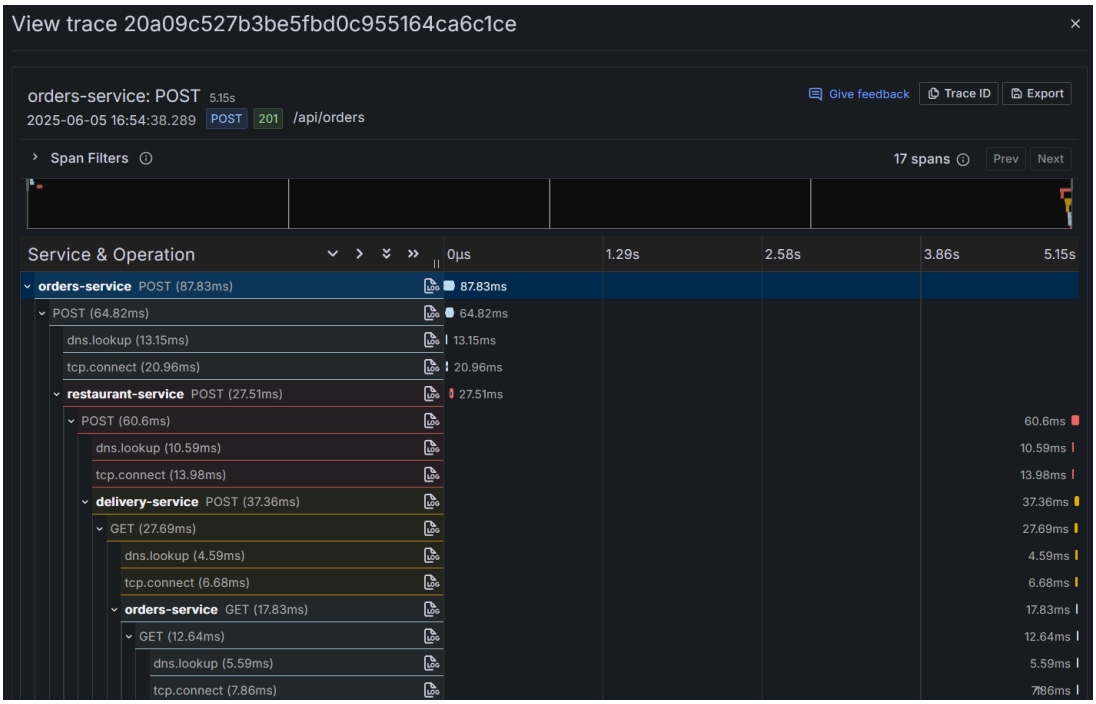
Rysunek 8.1: Architektura systemu do monitoringu projektu

8.4.1. Komponenty monitoringu

- **OpenTelemetry SDK** – instrumentacja aplikacji
- **OpenTelemetry Collector** – zbieranie telemetrii
- **Tempo** – przechowywanie traces
- **Prometheus** – metryki aplikacyjne i systemowe
- **Loki** – centralizowane logi
- **Grafana** – wizualizacja i dashboardy

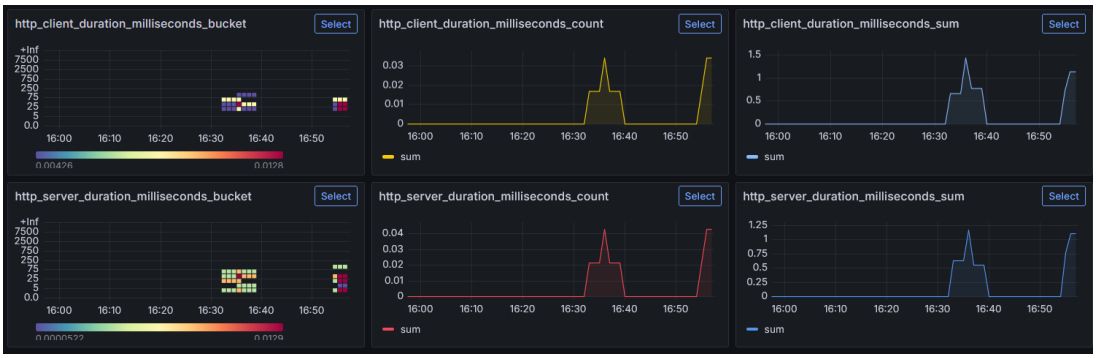
Wizualizacja takiej konfiguracji znajduje się na rysunku 8.1.

8.4.2. Obserwacja połączeń z wykorzystaniem Grafany i OpenTelemetry



Rysunek 8.2: Ślad komunikacji pomiędzy usługami zarejestrowany w Tempo

Na rysunku 8.2 widoczny jest pełen ślad żądania klienta do serwera, którego połączenie zostało zrealizowane z użyciem Network Service Mesh. Czas trwania oraz komponenty pośredniczące są widoczne w szczegółach trace'u.



Rysunek 8.3: Czasy odpowiedzi klienta i serwera

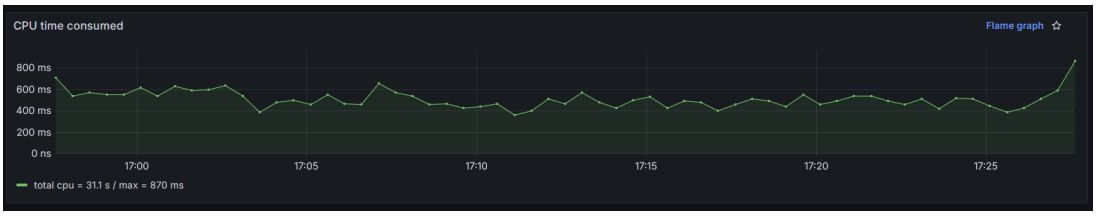
Rysunek 8.3 przedstawia metryki HTTP duration dla service mesh: rozkład czasów odpowiedzi (buckety), liczbę requestów oraz sumę czasów przetwarzania dla połączeń klient-serwer.

Rysunek 8.4: Metryki czasu odpowiedzi klienta HTTP

Rysunek 8.4 przedstawia metryki `http_client_duration`, które pokazują czas odpowiedzi dla żądań HTTP wychodzących z mikroserwisów. Widoczne są różnice w latencji między poszczególnymi serwisami, co pozwala na identyfikację potencjalnych problemów wydajnościowych.

Rysunek 8.5: Metryki czasu obsługi żądań HTTP po stronie serwera

Na rysunku 8.5 zaprezentowano metryki `http_server_duration`, które obrazują czas przetwarzania żądań HTTP przez poszczególne mikroserwisy. Dane te są kluczowe dla monitorowania wydajności aplikacji i identyfikacji usług wymagających optymalizacji.



Rysunek 8.6: Wykorzystanie czasu procesora

Rysunek 8.6 pokazuje wykorzystanie czasu procesora (`cpu_time`). Analiza tych danych pozwala na optymalne zarządzanie zasobami i planowanie skalowania aplikacji.

Rozdział 9

Użycie sztucznej inteligencji w projekcie

W projekcie sztuczna inteligencja została wykorzystana głównie jako wsparcie podczas debugowania. Narzędzia oparte na AI pomagały w szybkim lokalizowaniu błędów, analizie ich przyczyn oraz sugerowaniu możliwych rozwiązań. Dzięki temu znacznie ułatwiono proces diagnozowania problemów i poprawiania jakości kodu.

Rozdział 10

Podsumowanie - wnioski

Celem projektu było zaprojektowanie i przetestowanie architektury opartej na **Network Service Mesh (NSM)** w środowisku Kubernetes oraz ocena jej funkcjonalności w kontekście mikroserwisowej komunikacji i obserwowalności. W tym celu opracowano studium przypadku, obejmujące trzy współpracujące usługi (*Restaurant*, *Orders*, *Delivery*), które działały w odrębnych przestrzeniach nazw. Komunikacja między nimi realizowana była z wykorzystaniem NSM, co pozwoliło na dynamiczne zestawianie bezpiecznych połączeń sieciowych bez ingerencji w kod aplikacji.

W celu analizy działania systemu oraz wizualizacji danych telemetrycznych zastosowano narzędzie **OpenTelemetry (OTel)**, które umożliwiło kompleksowe zbieranie metryk, logów oraz śledzenie tras żądań (*tracing*) pomiędzy mikroserwisami. Zebrane dane były następnie wizualizowane w Grafanie, co umożliwiło szczegółową analizę wydajności i wykrywanie potencjalnych wąskich gardeł w komunikacji między usługami.

Network Service Mesh

- **Izolacja i bezpieczeństwo** – NSM umożliwia tworzenie w pełni izolowanych, dynamicznych połączeń między usługami, co podnosi bezpieczeństwo komunikacji i pozwala na granularną kontrolę nad ruchem sieciowym.

- **Brak potrzeby konfiguracji w kodzie aplikacji** – połączenia sieciowe definiowane są na poziomie deklaratywnym, niezależnie od implementacji aplikacji, co upraszcza wdrażanie i zmniejsza ryzyko błędów konfiguracyjnych.

- **Skalowalność** – podejście NSM dobrze skaluje się w środowiskach produkcyjnych, w których liczba usług dynamicznie się zmienia, a wymagana jest elastyczna kontrola nad ich połączeniami.

- **Integracja z narzędziami obserwowalności** – połączenie NSM z OpenTelemetry pozwala na śledzenie przepływu żądań nawet przez złożone topologie usług, co jest nieosiągalne przy klasycznym podejściu opartym na routingu IP.

OpenTelemetry

- **Kompletna telemetria** – OpenTelemetry zapewniło spójny sposób zbierania metryk, logów i śledzeń, co pozwoliło na całościowe spojrzenie na stan i wydajność systemu.

- **Elastyczność integracji** – OTel okazał się łatwy do integracji z istniejącymi narzędziami monitoringu (np. Grafana), co pozwoliło na szybką iterację i prezentację danych w czasie rzeczywistym.

Podsumowanie końcowe

Zrealizowany projekt pokazał praktyczne możliwości wykorzystania nowoczesnych narzędzi i podejść w budowie systemów mikroserwisowych działających w Kubernetes. W szczególności zastosowanie **Network Service Mesh** umożliwiło elastyczne zarządzanie komunikacją sieciową między usługami bez konieczności ingerencji w ich kod oraz z zachowaniem wysokiego poziomu izolacji i bezpieczeństwa. Z kolei wykorzystanie mechanizmów obserwowalności pozwoliło uzyskać szczegółowy wgląd w działanie systemu i skutecznie identyfikować potencjalne problemy wydajnościowe.

Efekty przeprowadzonych testów i obserwacji potwierdzają, że takie podejście dobrze sprawdza się w złożonych, dynamicznych środowiskach, gdzie istotna jest zarówno kontrola nad ruchem sieciowym, jak i bieżące monitorowanie działania usług. Uzyskane wyniki mogą stanowić wartościowy punkt odniesienia przy projektowaniu podobnych architektur w środowiskach produkcyjnych.

Spis rysunków

Spis listingów

Bibliografia

[1] *Network Service Mesh*. Dostęp: 2025-05-26. 2025. URL: https://networkservicemesh.io/.

[2] *Open Telemetry*. Dostęp: 2025-05-26. 2025. URL: https://opentelemetry.io/.