Hubble - OTel

Szymon Sadowski, Szymon Mamoń, Szczepan Rzeszutek, Jakub Wojciechowski

2025, grupa środa 15:00

1 Spis treści

2 Wprowadzenie

2.1 Teoria

2.1.1 Cilium

Dzięki wykorzystaniu technologii eBPF, eliminuje konieczność używania proxy w klastrach kubernetes. Większość CNI (CNI - z ang. Container Network Interaface, po polsku - Interfejs Sieciowy Kontenera) opiera się na iptables, czyli programu filtrującego pakiety poprzez tabele zawierające łańcuchy reguł stosowane dla pakietów. Iptables jest niestety słabo skalowalny, zwiększając opóźnienia i zmniejszając przepustowość. Dlatego Cilium korzysta z eBPF, co pozwala na lepszą skalowalność. Cilium pozwala również na load balancing na warstwie 4 oraz na tworzenie service/cluster mesha. Cilium implementuje również warstwę observability - Hubble. Cilium składa się z 4 głównych komponentów:

- Cilium operator
- Cilium agent
- Cilium CLI
- Cilium CNI plugin

Cilium operator zapewnia, że Cilium jest poprawnie skonfigurowany na każdym nodzie klastra. Cilium agent, daemon na każdym nodzie, zarządza programami eBPF, służącymi do kontroli uprawnień sieciowych kontenerów. Cilium CLI pozwala na konfigurację Cilium z terminala. CNI plugin integruje Cilium z kubernetesem, dzięki czemu klaster może wykorzystywać implementowane przez Cilium rozwiązania. Cilium pozwala na zarządzanie pakietami poprzez etykietowanie. Cilium pozwala także zaimplementować zero-trust security (wso2 case study).

2.1.2 eBPF

eBPF (Extended Berkeley Packet Filter) pozwala na wykonywanie kodu w jądrze Linuxa (lub innych systemów operacyjnych). Programy mogą reagować na dowolne wydarzenia w systemie. Pozwala to zbierać dane (observability). Poprzez eBPF można również wpływać na zachowanie jądra, np. przekierowując/modyfikując pakiety. Zmiany te mogą zachodzić dynamicznie. ### Hubble Observability na warstwie 3, 4 i 7. Pozwala się dowiedzieć: Jakie serwisy się ze sobą komunikują? Jak często? Jakie są między nimi zależności? Jakie HTTP calle są wysyłane? Z których tematów Kafka konsumuje serwis? Do których produkuje? Czy jakaś komunikacja sieciowa zawodzi? Czemu zawodzi? Czy jest to przez DNS? Czy jest to problem z aplikacją, czy z siecią? Na której warstwie jest problem (4 - TCP, czy 7 - HTTP)? Jak często występują kody 4xx lub 5xx HTTP dla któregoś serwisu albo na przestrzeni całego klastra? Jakie jest opóźnienie? Które serwisy mają zablokowaną komunikację (na podstawie zainstalowanych reguł)? Które serwisy mogą komunikować się poza klastrem?

2.1.3 Open Telemetry

Zestaw narzędzi do observability do zbierania danych takich jak ślady, metryki, logi oraz profile. Może być używany w połączeniu z wieloma innymi narzędziami, takimi, które są open-source (jak np. Jaeger) albo komercyjnymi. Open Telemetry jest wszechstronne, działa dla aplikacji lub systemu, niezależnie w jakim języku programowania jest on napisany, jaką ma infrastrukturę, czy środowisko wykonawcze. Open Telemetry składa się z kilku głównych komponentów: 1. API - konieczne, aby móc korzystać z narzędzia; 2. SDK - pozwala na szerszą konfigurację, na przykład filtrowanie zapytań, próbkowanie transakcji; 3. In-process exporter - jest częścią SDK. Pozwala skonfigurować, do którego backendu trafią dane telemetrii. Ułatwia zmianę używanego backendu; 4. Collector - Jest wielce użyteczny, ale nie jest koniecznym elementem OTel. Daje większą swobodę w otrzymywaniu, transformowaniu i wysyłaniu telemetrii do backendu. Jest niezależnym procesem, służącym jako centrum zbierania wszelkich danych telemetrii oraz ich przetwarzania.

2.2 Koncept demo

W celu zrealizowania projektu użyjemy demo aplikacji "Online Boutique" udostępnionej przez Google Cloud Platfrom. "Online Boutiqe" to sklep internetowy oparty na mikroserwisach, który umożliwia przeglądanie katalogu, dodawanie przedmiotów do koszyka oraz kupowanie ich. Pody klastra będą zarządzane przez Cilium. Aby było to możliwe, dodamy do manifestu klastra (plik .yaml) tzw. taint, czyli odpowiednią etykietę, umożliwiającą Cilium połączenie się z podem. Następnie za pomocą Hubble oraz OpenTelemetry będziemy obserwować ruch na klastrze i zbierać dane. W ramach projektu będziemy zbierać profile - zużycie CPU, pamięci, ślady - śledzenie ścieżki żądań oraz metryki - liczba żądań na sekundę, opóźnienie odpowiedzi. Otrzymane dane zwizualizujemy za pomocą Grafany. Na kolejnym etapie projektu uruchomimy taki sam sklep internetowy, ale korzystając z istio. Tutaj do wizualizacji danych będziemy używać Prometheus oraz Grafany. Cilium, które korzysta z eBPF powinno skalować się lepiej niż Istio, korzystające z iptables. Aby to sprawdzić porównamy opóźnienie odpowiedzie oraz użycie CPU oraz pamięci między implementacjami, przy takim samym ruchu.

2.3 Wykorzystane techonologie

- Cilium - do zapewnienia sieci, polityki bezpieczeństwa i load balancingu
- Hubble - do monitorowania i obserwacji ruchu sieciowego

- OTel - o wyciąganięcia metryk, trace'ów i log'ów
- Grafana - do wziualizacji

2.4 Konfiguracja środowiska

2.4.1 Jak uruchomić Cilium i Hubble (na minikube)

```
minikube start --driver=docker --cpus=2 --memory=4096 -p suu
```

```
helm repo add cilium https://helm.cilium.io/
```

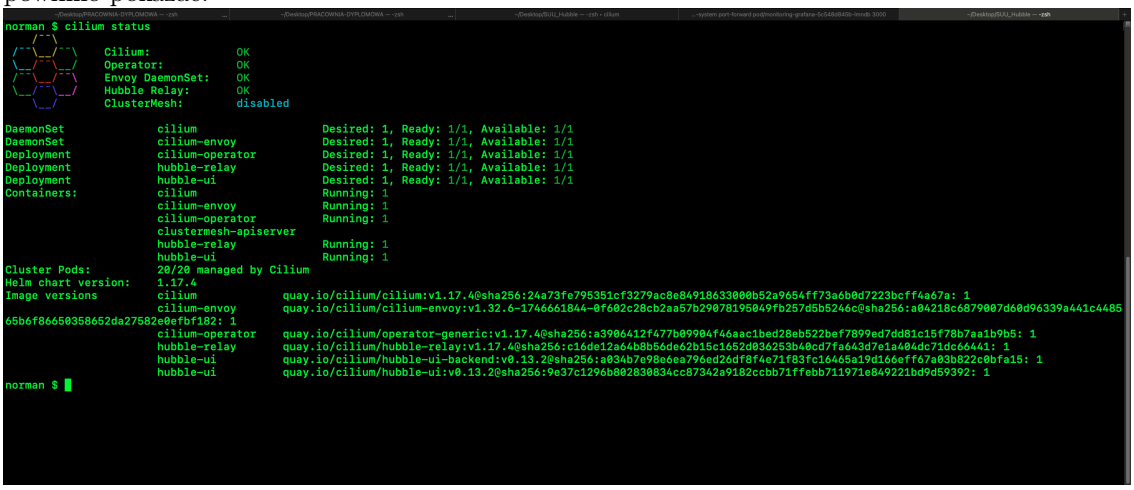
```
helm install cilium cilium/cilium --version 1.17.4 \
--namespace kube-system \
--set prometheus.enabled=true \
--set operator.prometheus.enabled=true \
--set hubble.enabled=true \
--set hubble.metrics.enableOpenMetrics=true \
--set hubble.metrics.enabled="{dns,drop,tcp,flow,port-
distribution,icmp,httpV2:exemplars=true;labelsContext=source_ip,
source_namespace,source_workload,destination_ip,destination_namespace,
destination_workload,traffic_direction}"
```

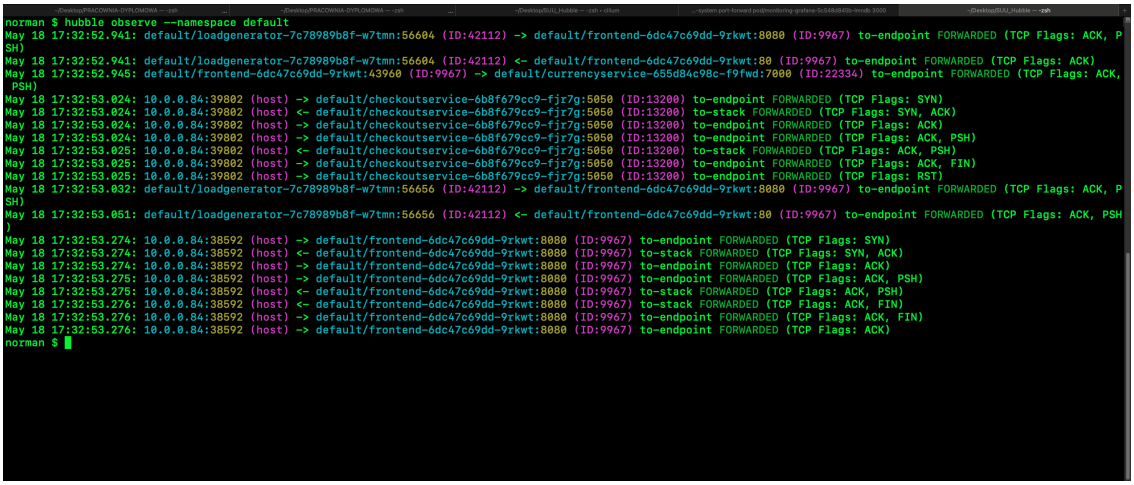
```
cilium hubble enable --ui
```

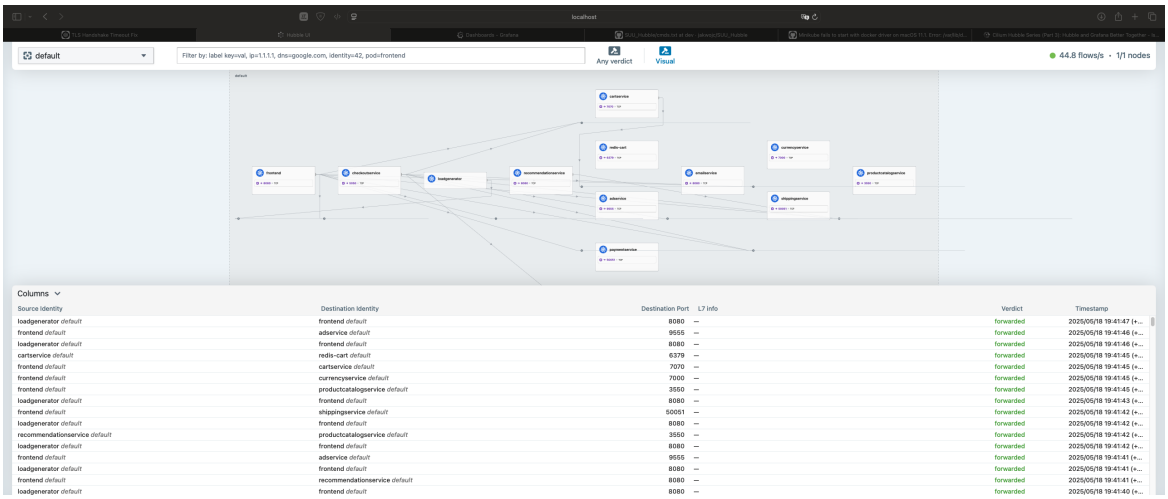
```
cilium hubble port-forward &
```

```
cilium status
```

powinno pokazać:





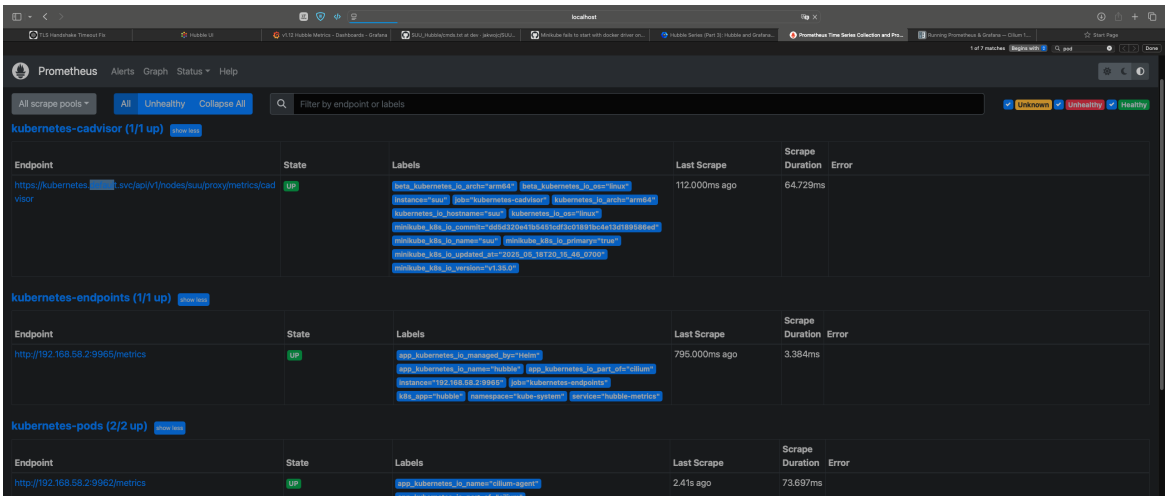



```
hubble observe --namespace default
```

```
cilium hubble ui
```

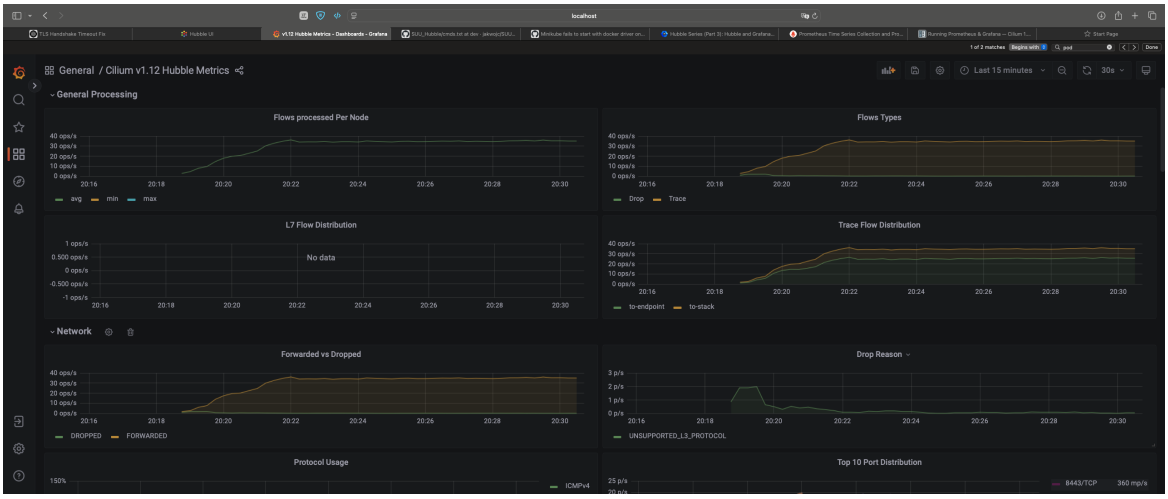
wejście na Prometheusa:

```
kubectl -n cilium-monitoring port-forward service/prometheus --address 0.0.0.0 --address :: 9090:9090
```



wejście na Grafane:

```
kubectl -n cilium-monitoring port-forward service/grafana --address 0.0.0.0 --address :: 3000:3000
```



zainstaluj cert-manager

```
kubectl apply -f https://github.com/cert-manager/cert-manager/releases/download/v1.17.2/cert-manager.yaml
```

zainstaluj opentelemtry operator

```
kubectl apply -f \
https://github.com/open-telemetry/opentelemetry-operator/releases/latest/download/opentelemetry-operator.yaml && kubectl apply -f otecol.yaml
```

w simplest-collector dodaj exportowanie metryk

```
kubectl edit opentelemetrycollector simplest
```

w nastepujacy sposob:

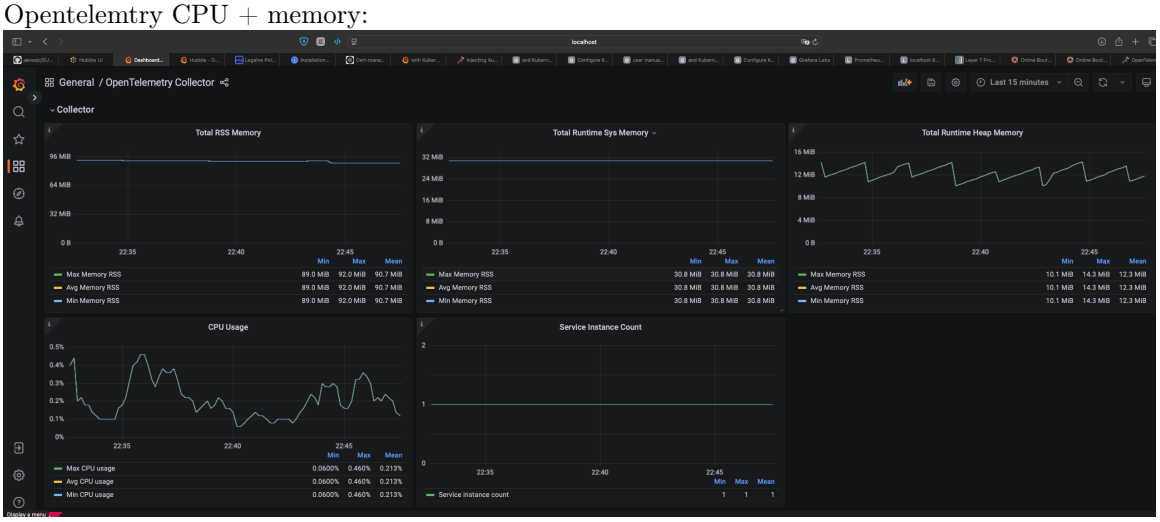
```
spec: config: exporters: debug: {} processors: batch: send_batch_size: 10000 timeout: 10s memory_limiter: check_interval: 1s limit_percentage: 75 spike_limit_percentage: 15 receivers: hostmetrics: scrapers: cpu: {} otlp: protocols: grpc: endpoint: 0.0.0.0:4317 http: endpoint: 0.0.0.0:4318 service: pipelines: metrics: exporters: - debug processors: - batch receivers: - hostmetrics traces: exporters: - debug processors: - memory_limiter - batch receivers: - otlp telemetry: metrics: readers: - pull: exporter: prometheus: host: 0.0.0.0 port: 8888
```

dodaj visibility na L7

```
kubectl apply -f ciliumnetworkpolicy.yaml
```

2.5 Grafana:

Opentelemtry CPU + memory:

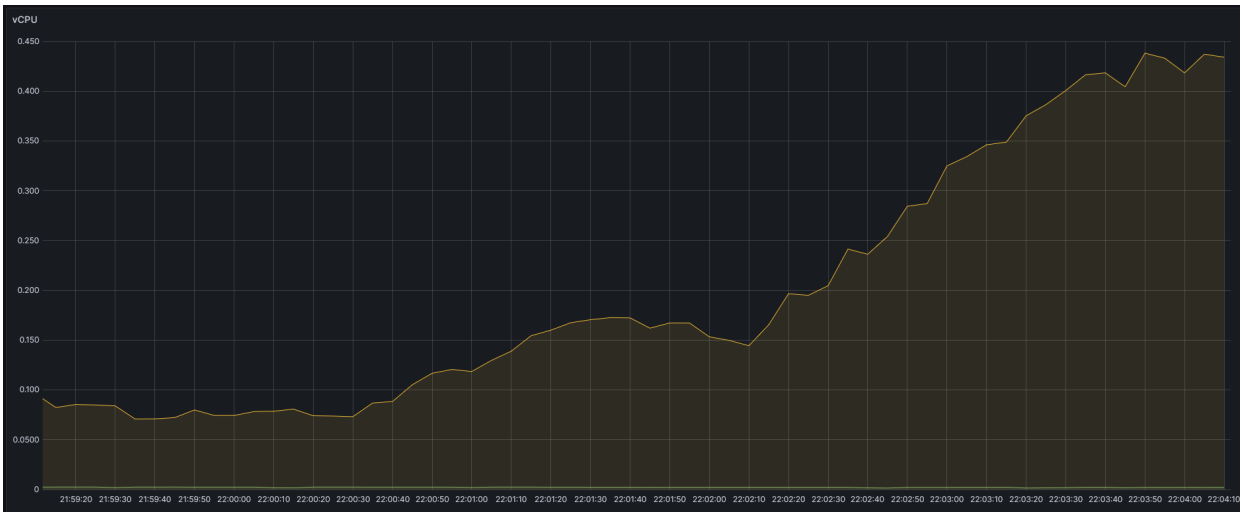


Hubble requests latency:

2.5.1 Uruchamianie na Istio

Demo na istio zostało uruchomione według instrukcji service mesh.

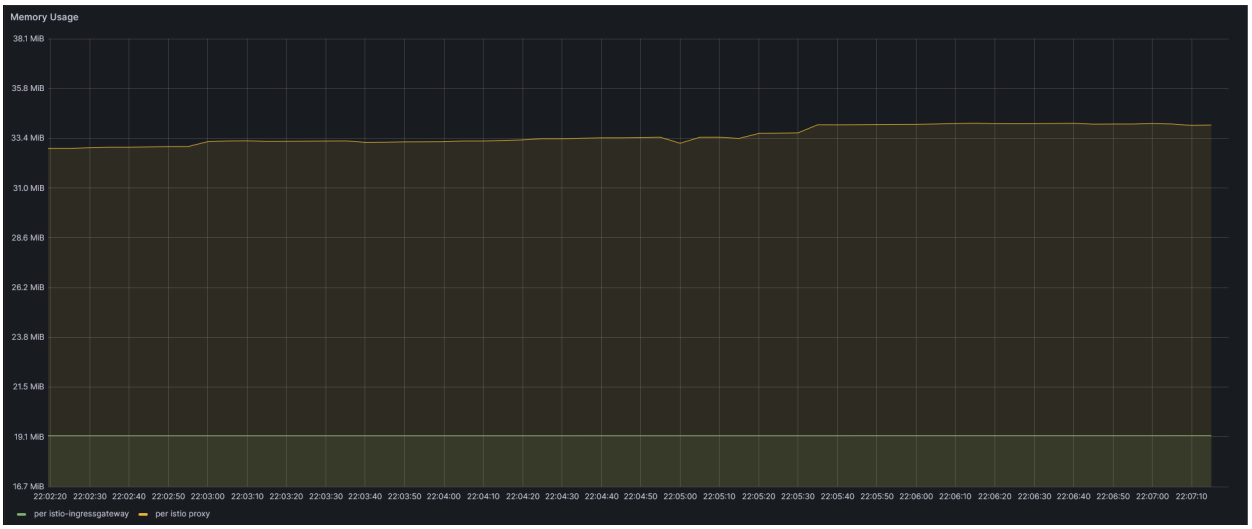
2.6 Porównanie z istio:



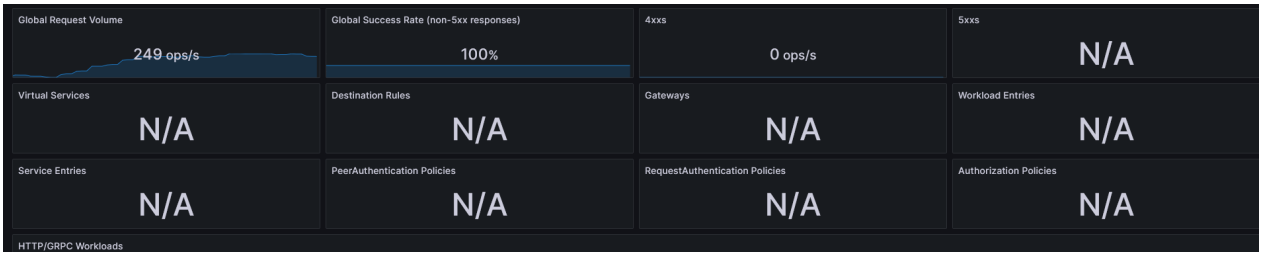
Rysunek 1: Istio - CPU

2.7 Podsumowanie

W ramach projektu wykorzystaliśmy Cilium do uruchomienia przykładowej mikroserwisowej aplikacji ‘Online boutique’. Zaimplementowanie Cilium na klastrze było proste - wystarczyło dodać



Rysunek 2: Istio - Pamięć

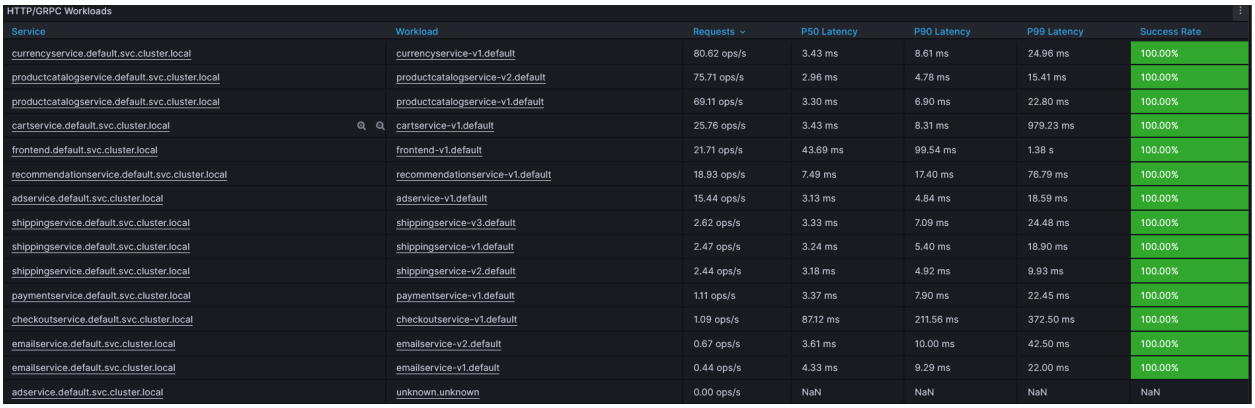


Rysunek 3: Istio - Ruch

odpowiednią etykietę w manifeście aplikacji oraz zainstalować sam program. Kolejnym celem projektu było wykorzystanie Hubble oraz Open Telemetry do observability na klastrze. Za pomocą Hubble udało nam się śledzić ruch żądań HTTP na klastrze (L7) oraz zmierzyć czas odpowiedzi na żądania. Hubble umożliwił również śledzenie ilości operacji wykonywanych na klastrze na sekundę. Wykorzystując Open Telemetry Collector zebraliśmy profile, takie jak wykorzystanie CPU oraz pamięci przez aplikację.

Architektura Cilium + Hubble różni się od standardowej implementacji wykorzystującej iptables. W przypadku kalstra zarządzanego przez Cilium wykorzystywana jest technologia eBPF. Pozwala to na znacznie wydajniejsze skalowania klastra niż w przypadku iptables. Ostatnim elementem projektu jest porównanie aplikacji 'Online Boutique' działającej na architekturze Cilium, z tą samą aplikacją zarządzaną przez Istio. Kluczowym faktem uzasadniającym takie porównanie jest to, że Istio wykorzystuje iptables, a Cilium - eBPF.

Z otrzymanych wyników niezaprzeczalnie widać przewagę Cilium nad Istio. Zwiększając ruch na klastrze, od 10 do 100 użytkowników, w przypadku Istio, wykorzystanie CPU rośnie znacząco - wręcz liniowo razem z liczbą użytkowników. W przypadku Cilium nie zauważono znaczącego wzrostu



HTTP/GRPC Workloads

Service	Workload	Requests	P50 Latency	P90 Latency	P99 Latency	Success Rate
currencyservice.default.svc.cluster.local	currencyservice-v1.default	80.62 ops/s	3.43 ms	8.61 ms	24.96 ms	100.00%
productcatalogservice.default.svc.cluster.local	productcatalogservice-v2.default	75.71 ops/s	2.96 ms	4.78 ms	15.41 ms	100.00%
productcatalogservice.default.svc.cluster.local	productcatalogservice-v1.default	69.11 ops/s	3.30 ms	6.90 ms	22.80 ms	100.00%
cartservice.default.svc.cluster.local	cartservice-v1.default	25.76 ops/s	3.43 ms	8.31 ms	979.23 ms	100.00%
frontend.default.svc.cluster.local	frontend-v1.default	21.71 ops/s	43.69 ms	99.54 ms	1.38 s	100.00%
recommendationservice.default.svc.cluster.local	recommendationservice-v1.default	18.93 ops/s	7.49 ms	17.40 ms	76.79 ms	100.00%
adservice.default.svc.cluster.local	adservice-v1.default	15.44 ops/s	3.13 ms	4.84 ms	18.59 ms	100.00%
shippingservice.default.svc.cluster.local	shippingservice-v3.default	2.62 ops/s	3.33 ms	7.09 ms	24.48 ms	100.00%
shippingservice.default.svc.cluster.local	shippingservice-v1.default	2.47 ops/s	3.24 ms	5.40 ms	18.90 ms	100.00%
shippingservice.default.svc.cluster.local	shippingservice-v2.default	2.44 ops/s	3.18 ms	4.92 ms	9.93 ms	100.00%
paymentservice.default.svc.cluster.local	paymentservice-v1.default	1.11 ops/s	3.37 ms	7.90 ms	22.45 ms	100.00%
checkoutservice.default.svc.cluster.local	checkoutservice-v1.default	1.09 ops/s	87.12 ms	211.56 ms	372.50 ms	100.00%
emailservice.default.svc.cluster.local	emailservice-v2.default	0.67 ops/s	3.61 ms	10.00 ms	42.50 ms	100.00%
emailservice.default.svc.cluster.local	emailservice-v1.default	0.44 ops/s	4.33 ms	9.29 ms	22.00 ms	100.00%
adservice.default.svc.cluster.local	unknown.unknown	0.00 ops/s	NaN	NaN	NaN	NaN

Rysunek 4: Istio - Opóźnienie

wykorzystania CPU. Następnie porównano czasy odpowiedzi na żądanie - przy ruchu generowanym przez 100 symulowanych użytkowników. Na Istio czas ten wyniósł około 1.3s w najgorszym wypadku, natomiast dla Cilium czas ten wyniósł nieco poniżej jednej sekundy. Wynika z tego, że Cilium pozwala na wydajniejsze skalowanie klastra niż Istio.

2.8 Referencje

1. Dokumentacja Cilium + Hubble
2. Open Telemetry Operator for Kubernetes
3. Hubble cheat-sheet
4. Dashboard OTel Collector w Grafanie