Zarządzanie Systemami Rozproszonymi
Laboratoria z Prometheus i Grafana

mgr inż. Jakub Woźniak

1 Wprowadzenie

W systemach rozproszonych monitoring stanowi istotne wyzwanie ze względu na złożoność środowiska i dynamikę jego zmian. W typowych produkcyjnych klastrach mikroserwisów aplikacje mogą skalować się horyzontalnie, co oznacza, że jedna usługa może działać na wielu instancjach jednocześnie. Monitorowanie pojedynczego Poda dostarcza jedynie wycinkowej informacji o stanie systemu, dlatego wymagane są narzędzia, które pozwolą zbierać, przetwarzać i agregować dane metryczne na poziomie całego klastra. Dzięki odpowiednim analizom statystycznym możliwe jest śledzenie zmian i reagowanie na potencjalne problemy w skali systemu.
`Prometheus` i `Grafana` są obecnie standardem monitorowania klastrów Kubernetes. Prometheus, jako system monitorowania i zbierania metryk, umożliwia elastyczne konfigurowanie reguł zbierania danych oraz definiowanie alertów. Grafana natomiast służy do wizualizacji i analizy zebranych danych, co pozwala na tworzenie intuicyjnych dashboardów prezentujących stan systemu w czasie rzeczywistym. Dzięki swojej wszechstronności narzędzia te są często obecne również w środowiskach, gdzie stosowane są komercyjne rozwiązania monitorujące – integrują się one z Prometheus i Grafana, oferując bardziej dedykowane rozwiązania dla monitorowania kontenerów i mikroserwisów.

1.1 Cel

Celem zajęć jest przybliżenie studentom praktyk związanych z monitorowaniem klastra oraz analizą metryk systemowych i aplikacyjnych. Uczestnicy zajęć nauczą się konfigurować Prometheus oraz Grafana w środowisku Kubernetes, a także integrować je z własną aplikacją, aby monitorować jej stan. Podczas ćwiczeń zostanie wykorzystana wiedza zdobyta na poprzednich zajęciach z Docker i Kubernetes.

1.2 Przygotowanie środowiska

Do przeprowadzenia ćwiczeń wymagany jest klaster Kubernetes, zainstalowany przy pomocy `minikube`. Instrukcja instalacji oraz konfiguracji klastra znajduje się w skrypcie do poprzednich laboratoriów z Kubernetes. Upewnij się,

że Minikube działa poprawnie, a dostęp do klastra można uzyskać za pomocą polecenia `kubectl`. Prometheus i Grafana zostaną zainstalowane jako aplikacje wewnątrz klastra przy użyciu Helm.

2 Wprowadzenie do Helma

`Helm` to popularny menedżer pakietów dla Kubernetes, który pozwala na instalację, aktualizację i zarządzanie aplikacjami w klastrze Kubernetes. Helm wykorzystuje tzw. „chart" – paczkę zawierającą wszystkie zasoby i konfiguracje niezbędne do wdrożenia aplikacji. Dzięki Helm można zarządzać bardziej złożonymi aplikacjami i ich zależnościami w jednym pliku konfiguracji.

2.1 Podstawowe polecenia Helm

- `helm repo add [nazwa] [adres_repozytorium]` – dodaje nowe repozytorium wykresów Helm, np. repozytorium Prometheus.
- `helm repo update` – aktualizuje listę wykresów w repozytorium.
- `helm install [nazwa_instalacji] [repozytorium/wykres] --namespace [namespace]` – instaluje aplikację w klastrze Kubernetes. Przykład: `helm install grafana prometheus-community/grafana --namespace monitoring`.
- `helm uninstall [nazwa_instalacji] --namespace [namespace]` – usuwa zainstalowaną aplikację z klastra.

2.2 Struktura wykresu Helm

Każdy wykres Helm (chart) składa się z następujących elementów:

- **Chart.yaml** – główny plik wykresu zawierający informacje o wersji aplikacji, nazwie, opisie i zależnościach.
- **values.yaml** – plik konfiguracyjny, gdzie można zmieniać domyślne wartości konfiguracji aplikacji.
- **templates/** – katalog zawierający szablony YAML dla Kubernetes, które Helm przekształca na pliki konfiguracyjne dla klastra podczas wdrożenia.

Helm ułatwia instalację i zarządzanie aplikacjami wieloskładnikowymi, automatyzując tworzenie, aktualizację i zarządzanie konfiguracją wielu zasobów Kubernetes w jednym pakiecie.

3 Zadania

3.1 Instalacja Helm

1. Zainstaluj Helm, jeśli jeszcze tego nie zrobiłeś, korzystając z instrukcji dostępnych na stronie projektu Helm: `https://helm.sh/docs/intro/install/`.

2. Zaktualizuj repozytoria Helm, aby pobrać najnowsze wersje wykresów Helm:

```
helm repo add prometheus-community
    https://prometheus-community.github.io/helm-charts
helm repo add grafana
    https://grafana.github.io/helm-charts
helm repo update
```

3.2 Instalacja Prometheus i Grafana

1. Użyj Helm, aby zainstalować Prometheus i Grafana na klastrze:

```
helm install prometheus
    prometheus-community/prometheus --namespace
    monitoring --create-namespace
helm install grafana grafana/grafana --namespace
    monitoring
```

2. Po instalacji uzyskaj hasło administratora dla Grafana, używając poniższego polecenia:

```
kubectl get secret --namespace monitoring grafana
    -o jsonpath="{.data.admin-password}" | base64
    --decode; echo
```

3. Uzyskaj dostęp do interfejsu Grafana, przekierowując port:

```
kubectl port-forward --namespace monitoring
    service/grafana 3000:80
```

3.2.1 Wyjaśnienie polecenia `port-forward`

Polecenie `port-forward` służy do przekierowania portu z klastra Kubernetes na lokalny komputer, co pozwala na dostęp do usług działających w klastrze bez potrzeby wystawiania ich na zewnątrz. W powyższym przykładzie polecenie:

```
kubectl port-forward --namespace monitoring
    service/grafana 3000:80
```

przekierowuje ruch z portu `3000` na lokalnej maszynie na port `80` usługi `grafana` działającej w klastrze w przestrzeni nazw `monitoring`. Dzięki temu możemy uzyskać dostęp do interfejsu Grafana poprzez `http://localhost:3000`.

3.3 Instalacja node-exporter i serwera metryk

Aby monitorować klaster Kubernetes, potrzebujemy dostępu do podstawowych metryk dotyczących zasobów systemowych oraz stanu węzłów i podów. W tym celu wykorzystamy narzędzia takie jak `node-exporter` oraz `metrics-server`.
`node-exporter` to komponent Prometheus, który umożliwia zbieranie szczegółowych metryk systemowych bezpośrednio z węzłów Kubernetes. Zbierane metryki obejmują informacje o zużyciu CPU, pamięci, obciążeniu sieci, a także statystyki dyskowe. Dzięki `node-exporter` możemy śledzić stan infrastruktury klastra i szybko reagować na problemy związane z zasobami systemowymi. Jest to szczególnie przydatne w środowiskach rozproszonych, gdzie monitorowanie zasobów systemowych ma kluczowe znaczenie dla stabilności aplikacji.
`metrics-server` to dodatek Kubernetes, który dostarcza metryki dotyczące zasobów na poziomie podów oraz węzłów. Jest wykorzystywany m.in. do realizacji horyzontalnego autoskalowania (HPA) i do monitorowania zużycia zasobów przez aplikacje. `metrics-server` jest lekkim serwerem, który dostarcza podstawowe metryki klastra, takie jak zużycie procesora i pamięci przez poszczególne pody oraz węzły. Jest to standardowy komponent w klastrach Kubernetes, który umożliwia użytkownikom analizowanie podstawowych wskaźników wydajności klastra.

3.3.1 Instalacja metrics-server

Aby włączyć `metrics-server` w klastrze Kubernetes przy pomocy Minikube, wykonaj następujące polecenie:

```
minikube addons enable metrics-server
```

Po włączeniu `metrics-server` klaster będzie zbierał metryki o zużyciu procesora i pamięci przez pody oraz węzły. Metryki te będą dostępne do przeglądania i analizy, a także mogą być używane do skalowania aplikacji w klastrze.

3.3.2 Instalacja node-exporter

Aby zainstalować `node-exporter` w klastrze, wykonaj instalację `node-exporter` przy pomocy Helm w przestrzeni nazw `monitoring`:

```
helm install node-exporter
    prometheus-community/prometheus-node-exporter
    --namespace monitoring --create-namespace
```

Po zakończeniu instalacji `node-exporter` zacznie zbierać i udostępniać metryki systemowe węzłów. Dzięki integracji z Prometheus, te metryki będą dostępne do analizy w Grafana, co pozwoli na bardziej kompleksowy monitoring zasobów klastra.

3.4 Generowanie sztucznego obciążenia w klastrze Kubernetes

Aby przeprowadzić testy monitorowania i zobaczyć reakcje systemu na zwiększone zużycie zasobów, wykorzystamy narzędzie `stress-ng`. Jest to popularne narzędzie do generowania sztucznego obciążenia procesora, pamięci oraz innych zasobów systemowych. Umożliwia symulację różnych scenariuszy obciążeniowych, co pozwala na testowanie wydajności klastra oraz efektywności zbierania i analizowania metryk.
W naszym laboratorium użyjemy obrazu `polinux/stress-ng`, który został przygotowany specjalnie do testowania wydajności w środowiskach kontenerowych. Dzięki `stress-ng` będziemy mogli wygenerować różnorodne obciążenie, które pozwoli nam na zweryfikowanie działania konfiguracji monitoringu oraz obserwację wpływu obciążenia na metryki zbierane przez Prometheus i wyświetlane w Grafana.
Przykład zastosowania
W poniższych przykładach stworzymy dwa różne Deployments:
1. Pierwszy Deployment będzie generował obciążenie procesora (CPU).
2. Drugi Deployment będzie obciążał pamięć.

Dzięki zastosowaniu wielu replik, oba deployments będą generować istotne obciążenie na poziomie klastra.

3.4.1 Generowanie obciążenia CPU

W pierwszym przykładzie stwórzmy Deployment z kilkoma replikami, które będą generować obciążenie na procesorze.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cpu-stress-test
spec:
  replicas: 3
  selector:
    matchLabels:
      app: cpu-stress
  template:
    metadata:
      labels:
        app: cpu-stress
    spec:
      containers:
      - name: stress-ng
        image: polinux/stress
        command: ["stress", "--cpu", "4", "--timeout",
            "600s"]
```

W powyższym przykładzie:

- Ustawiliśmy trzy repliki dla Deploymentu, aby wygenerować skumulowane obciążenie na CPU.
- Każdy kontener wykonuje polecenie ‘stress –cpu 4 –timeout 600s‘, co oznacza, że generuje obciążenie na 4 rdzeniach procesora przez 10 minut.

3.4.2 Generowanie obciążenia pamięci

W drugim przykładzie skonfigurujemy Deployment, który obciąży pamięć klastra.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: memory-stress-test
spec:
  replicas: 3
  selector:
    matchLabels:
      app: memory-stress
  template:
    metadata:
      labels:
        app: memory-stress
    spec:
      containers:
      - name: stress-ng
        image: polinux/stress
        command: ["stress", "--vm", "2", "--vm-bytes",
            "512M", "--timeout", "600s"]
```

W powyższym przykładzie:

- Również używamy trzech replik.
- Każdy kontener wykonuje polecenie ‘stress –vm 2 –vm-bytes 512M –timeout 600s‘, które generuje obciążenie pamięci, tworząc dwa procesy zajmujące po 512 MB każdy przez 10 minut.

Oba powyższe Deploymenty pozwolą wygenerować sztuczne obciążenie, które może być monitorowane przez system Prometheus i wizualizowane w Grafana. Dzięki temu będziemy mogli zaobserwować zmiany w zużyciu CPU i pamięci w klastrze oraz zweryfikować poprawność konfiguracji monitoringu.

3.5 Wdrożenie NGINX z metrykami i generowanie ruchu przy użyciu Apache Benchmark

Aby wygenerować rzeczywiste dane metryczne, wdrożymy serwer NGINX, który obsługuje żądania HTTP. Dodatkowo zainstalujemy eksporter metryk dla NGINX, aby Prometheus mógł monitorować wydajność serwera, a następnie użyjemy narzędzia `Apache Benchmark` (`ab`) w osobnym podzie, aby wygenerować ruch.

3.5.1 Wdrożenie serwera NGINX z eksportem metryk

Eksporter metryk NGINX pozwala Prometheus na zbieranie danych, takich jak liczba obsługiwanych żądań, czas odpowiedzi, oraz wykorzystanie zasobów. Zastosujemy dwa kontenery w jednym podzie: jeden z NGINX oraz drugi z eksporterem metryk dla NGINX.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
      annotations:
        prometheus.io/scrape: "true"
        prometheus.io/port: "9113"
        prometheus.io/path: "/metrics"
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
        volumeMounts:
        - mountPath: /etc/nginx/conf.d
          name: config
      - name: nginx-exporter
        image: nginx/nginx-prometheus-exporter
        args:
        -
            -nginx.scrape-uri=http://127.0.0.1:80/stub_status
```

```
        ports:
        - containerPort: 9113
      volumes:
      - name: config
        configMap:
          name: nginx-config
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-config
data:
  default.conf: |
    server {
        listen 80;
        location / {
            root /usr/share/nginx/html;
        }
        location /stub_status {
            stub_status on;
            access_log off;
            allow all;
        }
    }
---
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

W tym przykładzie dodaliśmy trzy kluczowe adnotacje do metadanych szablonu podów NGINX:

- `prometheus.io/scrape: "true"`: informuje Prometheus, że pod powinien być monitorowany.
- `prometheus.io/port: "9113"`: określa port, na którym Prometheus powinien zbierać metryki (w tym przypadku port eksportera NGINX).

- `prometheus.io/path: "/metrics"`: definiuje ścieżkę, z której Prometheus pobiera metryki.

Te adnotacje umożliwiają Prometheus automatyczne wykrywanie eksportera metryk na kontenerze NGINX, dzięki czemu nie ma potrzeby ręcznego modyfikowania konfiguracji `prometheus.yml`. Metryki z serwera NGINX powinny być teraz automatycznie dostępne do monitorowania i wizualizacji w Prometheus oraz Grafana.

Kontener Sidecar to dodatkowy kontener uruchomiony w tym samym Podzie co główna aplikacja. Działa jako pomocniczy proces wspierający funkcjonalność głównej aplikacji. W Kubernetes sidecar jest używany m.in. do zbierania metryk, logów lub do innych zadań pomocniczych, które wspierają działanie głównego kontenera.
Ponieważ kontenery działające w jednym Podzie współdzielą zasoby, takie jak sieć i system plików (przez współdzielone wolumeny), sidecar ma bezpośredni dostęp do tych zasobów. To umożliwia łatwe monitorowanie i współpracę między kontenerami. Przykładem sidecara jest `PostgreSQL Exporter`, który działa obok bazy danych PostgreSQL i zbiera jej metryki.
W powyższym przykładzie kontener `nginx-exporter` łączy się po `localhost` (ze względu na współdzielenie interfejsu) z głównym kontenerem `nginx` w celu pobrania metryk systemowych. Następnie sam wystawia te metryki na innym porcie, już w formacie zgodnym z `Prometheus`. **Sidecar** to popularny wzorzec projektowy w kontekście kontenerów, który pozwala na łatwe rozszerzanie funkcjonalności aplikacji poprzez dodawanie dodatkowych kontenerów w jednym Podzie.

3.5.2 Wdrożenie Apache Benchmark jako generatora ruchu

Aby wygenerować ruch do serwera NGINX, wdrożymy prosty pod z Apache Benchmark. Narzędzie `ab` będzie cyklicznie wysyłać żądania do NGINX, generując obciążenie, które możemy monitorować w Prometheus.

```
apiVersion: v1
kind: Pod
metadata:
  name: apache-benchmark
spec:
  containers:
  - name: apache-benchmark
    image: jordi/ab
    command: ["sh", "-c", "while true; do ab -n 100 -c
       10 http://nginx.default.svc.cluster.local/; done"]
```

W powyższym przykładzie:

- tworzymy pod z narzędziem `Apache Benchmark`, które wykonuje cykliczne testy obciążeniowe.

- `ab` wysyła 100 żądań z 10 jednoczesnymi połączeniami do usługi NGINX dostępnej w klastrze Kubernetes.

Podczas uruchomienia tego scenariusza Prometheus będzie zbierać metryki z serwera NGINX, a w Grafana będziemy mogli obserwować dane dotyczące wydajności serwera, w tym liczbę obsłużonych żądań oraz wykorzystanie zasobów. Dzięki temu uzyskamy pełne środowisko monitorowania z symulowanym ruchem i rzeczywistymi danymi.

4 Korzystanie z Prometheus do monitorowania metryk

Prometheus to zaawansowane narzędzie do monitorowania metryk w środowisku Kubernetes. Działa na zasadzie time-series, co oznacza, że dane zbierane są jako szereg czasowy, dzięki czemu można je analizować na bieżąco i śledzić trendy historyczne. Prometheus oferuje również język zapytań PromQL, który pozwala na analizę i agregację danych metrycznych.

4.1 Dostęp do interfejsu Prometheus

Aby uzyskać dostęp do interfejsu Prometheus, należy użyć polecenia `port-forward`, aby przekierować porty na lokalny komputer:

```
kubectl port-forward --namespace monitoring
    service/prometheus-server 9090:80
```

Po wykonaniu tego polecenia można otworzyć interfejs Prometheus w przeglądarce, przechodząc pod adres `http://localhost:9090`. Interfejs umożliwia wykonanie zapytań w języku PromQL oraz przeglądanie wyników w formie tabelarycznej lub graficznej.

4.2 Przykłady zapytań PromQL

Poniżej znajdują się przykłady zapytań, które można wykonać w Prometheus, aby analizować metryki generowane przez nasze usługi (NGINX oraz `stress-ng`).

1. **Sprawdzenie stanu monitorowanych instancji**: Wyświetla wszystkie monitorowane instancje oraz informację, czy są one dostępne (1 - dostępne, 0 - niedostępne).

   ```
   up
   ```

2. **Średnie obciążenie CPU generowane przez `stress-ng`**: Wyświetla średnie zużycie CPU na wszystkich węzłach w ciągu ostatnich 5 minut.

   ```
   avg(rate(node_cpu_seconds_total{mode!="idle"}[5m]))
   ```

To zapytanie wyświetla średnie obciążenie CPU, pomijając tryb bezczynności (idle), aby uzyskać lepszy obraz faktycznego obciążenia.

3. **Liczba żądań HTTP do serwera NGINX**: Wyświetla całkowitą liczbę żądań HTTP obsłużonych przez serwer NGINX, co pozwala monitorować ruch generowany przez Apache Benchmark.

   ```
   sum(nginx_http_requests_total)
   ```

4.3 Zadania samodzielne

Poniższe zadania pomogą Ci lepiej zrozumieć język zapytań PromQL oraz analizę metryk zbieranych przez Prometheus.

1. **Oblicz sumaryczne obciążenie CPU na wszystkich węzłach klastra**: Stwórz zapytanie, które wyświetli łączne zużycie CPU przez wszystkie węzły klastra, z wyłączeniem trybu bezczynności.
2. **Oblicz średnią liczbę obsługiwanych połączeń HTTP na sekundę przez NGINX**: Użyj odpowiedniego zapytania, aby obliczyć średnią liczbę połączeń HTTP obsługiwanych przez serwer NGINX na sekundę w ciągu ostatnich 5 minut.

Dokumentacja dotycząca języka zapytań Prometheus (PromQL) jest dostępna pod adresem: `https://prometheus.io/docs/prometheus/latest/querying/basics/`

5 Korzystanie z Grafana do wizualizacji metryk

`Grafana` to narzędzie służące do wizualizacji danych i monitorowania w czasie rzeczywistym, szczególnie przydatne w połączeniu z Prometheus. Dzięki Grafana można tworzyć intuicyjne i interaktywne dashboardy, które pozwalają na lepsze zrozumienie stanu systemu oraz jego metryk.

5.1 Dostęp do interfejsu Grafana

Aby uzyskać dostęp do interfejsu Grafana, należy przekierować port na lokalny komputer, podobnie jak w przypadku Prometheus:

```
kubectl port-forward --namespace monitoring
    service/grafana 3000:80
```

Po wykonaniu tego polecenia, interfejs Grafana będzie dostępny pod adresem `http://localhost:3000`. Domyślny login to `admin`, a hasło można pobrać za pomocą komendy:

```
kubectl get secret --namespace monitoring grafana -o
    jsonpath="{.data.admin-password}" | base64 --decode;
    echo
```

5.2 Konfiguracja Prometheus jako źródła danych

Aby Grafana mogła wizualizować dane z Prometheus, konieczne jest dodanie Prometheus jako źródła danych:

1. Przejdź do interfejsu Grafana i wybierz opcję **Configuration** (ikona trybika) w menu bocznym, a następnie kliknij **Data sources**.
2. Kliknij przycisk **Add data source**.
3. Wybierz `Prometheus` z listy dostępnych źródeł.
4. W polu **URL** wpisz `http://prometheus-server.monitoring.svc.cluster.local:80` (to wewnętrzny adres URL serwera Prometheus w klastrze Kubernetes).
5. Zatwierdź konfigurację, klikając **Save & Test**, aby upewnić się, że Grafana poprawnie połączyła się z Prometheus.

5.3 Tworzenie własnych dashboardów

Grafana pozwala użytkownikom na dużą swobodę w tworzeniu dashboardów, dzięki czemu mogą dostosować sposób wyświetlania metryk do swoich potrzeb. Poniżej znajdują się podstawowe kroki oraz wytyczne do samodzielnego skonfigurowania dashboardów:

1. W interfejsie Grafana wybierz opcję **Create** (ikona plusa) w menu bocznym, a następnie kliknij **Dashboard**.
2. Dodaj nowy panel klikając **Add new panel**.
3. W polu **Metrics** wpisz zapytania PromQL, korzystając z metryk poznanych w poprzednich sekcjach. Przykładowe metryki to:
   - `sum(nginx_http_requests_total)` – całkowita liczba żądań HTTP obsłużonych przez NGINX.
   - `avg(rate(node_cpu_seconds_total{mode!="idle"}[5m]))` – średnie zużycie CPU przez pody.
4. Eksperymentuj z różnymi typami wizualizacji (np. linia, wykres słupkowy, wskaźnik) oraz opcjami dostosowania wyglądu paneli, aby zobaczyć, jak najlepiej przedstawić dane.

5.4 Zadania samodzielne

W celu zdobycia doświadczenia w pracy z Grafana, studenci powinni samodzielnie stworzyć przynajmniej trzy różne panele na jednym dashboardzie, używając poznanych metryk.
Przykładowe zadania:

- Stwórz panel wyświetlający średnie obciążenie CPU dla wszystkich węzłów klastra.
- Skonfiguruj wskaźnik, który pokazuje bieżącą dostępność pamięci na wybranym węźle.
- Stwórz wykres pokazujący liczbę obsłużonych żądań HTTP przez NGINX w czasie.
- Spróbuj stworzyć wykresy liczby Podów w systemie.

Grafana to doskonałe narzędzie zapewniające obserwowalność klastra. Spróbuj modyfikować liczbę replik `nginx`, parametry `ab` i `stress-ng`, aby zobaczyć, jak zmiany wpływają na metryki i wizualizacje w Grafanie.

6 Konfiguracja Alertmanager i MailHog do monitorowania alertów

`Alertmanager` jest integralną częścią ekosystemu Prometheus, odpowiedzialną za zarządzanie alertami oraz ich eskalację. Pozwala na wysyłanie powiadomień w odpowiedzi na wcześniej zdefiniowane alerty, co umożliwia szybkie reagowanie na kluczowe zdarzenia w systemie. `Alertmanager` wspiera integracje z popularnymi systemami powiadomień, takimi jak Slack, e-mail, PagerDuty, oraz różne kanały webhooks, co czyni go elastycznym i przydatnym narzędziem w środowiskach produkcyjnych.
W tej sekcji skonfigurujemy Alertmanager, aby wysyłał powiadomienia przez email przy użyciu narzędzia `MailHog`. MailHog działa jako serwer SMTP, przechwytując wszystkie wysłane maile i wyświetlając je w przeglądarce. Dzięki temu możemy bezpiecznie testować konfigurację powiadomień, bez potrzeby wysyłania alertów do prawdziwego serwera pocztowego.

6.1 Instalacja Alertmanager

Alertmanager powinien już być zainstalowany na klastrze. Upewnij się, sprawdzając to poleceniem:

```
kubectl get statefulsets -n monitoring
```

Powyższe polecenie powinno zwrócić informację o `StatefulSets` w przestrzeni nazw `monitoring`. Jeżeli znajduje się tam wpis o `alertmanager` - wszystko działa prawidłowo. Możesz przejść do przekierowania portu:

```
kubectl port-forward --namespace monitoring
   service/alertmanager 9093:9093
```

Interfejs Alertmanager będzie dostępny pod adresem `http://localhost:9093`. W tym interfejsie można przeglądać aktywne alerty, zarządzać nimi oraz weryfikować wysyłane powiadomienia.

6.2 Instalacja MailHog

`MailHog` to narzędzie przeznaczone do przechwytywania i wyświetlania wiadomości e-mail. Działa jako lokalny serwer SMTP, który przechowuje wszystkie otrzymane wiadomości i udostępnia je w prostym interfejsie webowym. MailHog jest szczególnie przydatny do testowania systemów powiadomień, ponieważ umożliwia przechwycenie maili bez wysyłania ich na prawdziwe adresy email.
W celu instalacji MailHog w klastrze Kubernetes, użyjemy następującej definicji Deploymentu oraz Service:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mailhog
  namespace: monitoring
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mailhog
  template:
    metadata:
      labels:
        app: mailhog
    spec:
      containers:
      - name: mailhog
        image: mailhog/mailhog
        ports:
        - containerPort: 1025 # SMTP port
        - containerPort: 8025 # HTTP UI port
---
apiVersion: v1
kind: Service
metadata:
  name: mailhog
  namespace: monitoring
spec:
  selector:
    app: mailhog
  ports:
    - name: smtp
      protocol: TCP
      port: 1025
      targetPort: 1025
    - name: http
      protocol: TCP
```

```
    port: 8025
    targetPort: 8025
```

Po wdrożeniu MailHog można uzyskać dostęp do interfejsu webowego przez przekierowanie portu:

```
kubectl port-forward --namespace monitoring
   service/mailhog 8025:8025
```

Interfejs MailHog będzie dostępny pod adresem `http://localhost:8025`, gdzie wszystkie e-maile wysyłane przez Alertmanager będą przechwytywane i wyświetlane.

6.3 Konfiguracja Alertmanager do pracy z MailHog

Aby Alertmanager mógł wysyłać powiadomienia email do MailHog, zaktualizujemy konfigurację Alertmanager, tak aby wszystkie alerty były kierowane do serwera SMTP MailHog.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: alertmanager-config
  namespace: monitoring
data:
  alertmanager.yml: |
    route:
      receiver: 'email-notifications'
    receivers:
      - name: 'email-notifications'
        email_configs:
          - to: 'student@example.com'
            from: 'alertmanager@example.com'
            smarthost:
                'mailhog.monitoring.svc.cluster.local:1025'
            headers:
              Subject: "Alert: {{
                  .CommonLabels.alertname }}"
```

W tej konfiguracji:

- `route` kieruje wszystkie alerty do odbiorcy `email-notifications`.
- `email_configs` definiuje konfigurację SMTP dla MailHog:
  - `to` jest adresem email, na który Alertmanager wysyła powiadomienia (w przypadku MailHog nie ma to znaczenia, ale warto dodać dla spójności).
  - `smarthost` wskazuje na usługę MailHog w klastrze Kubernetes, która przechwytuje wiadomości email.

– `headers` pozwala na ustawienie niestandardowego tematu wiadomości.

Po zapisaniu konfiguracji zaktualizuj ConfigMap w klastrze:

```
kubectl apply -f alertmanager-config.yaml
kubectl rollout restart statefulset alertmanager -n
    monitoring
```

6.4 Tworzenie reguł alertów w Prometheus

Aby Alertmanager mógł reagować na określone zdarzenia, musimy zdefiniować reguły alertów w Prometheus. Poniżej znajdują się dwa przykłady alertów oraz instrukcje wywołania zdarzeń dla tych alertów.

6.4.1 Dodawanie alertów w Prometheus zainstalowanym przez Helm

Aby dodać alerty w Prometheus zainstalowanym za pomocą Helm, należy skonfigurować reguły alertów i dodać je do pliku wartości (`values.yaml`). Poniżej znajdują się szczegółowe instrukcje.

1. **Znajdź i edytuj plik `values.yaml`**
   Jeżeli masz lokalną kopię pliku `values.yaml` używanego do instalacji Prometheus przez Helm, możesz go edytować bezpośrednio. Jeśli nie, pobierz domyślny plik wartości z repozytorium Helm Prometheus:

   ```
   helm show values prometheus-community/prometheus
       > values.yaml
   ```

2. **Dodaj reguły alertów do pliku `values.yaml`**
   W pliku `values.yaml` znajdź sekcję `serverFiles` i dodaj własne reguły alertów do pliku `alerting_rules.yml`. Poniżej znajduje się przykładowa konfiguracja dla monitorowania wysokiego zużycia CPU:

   ```
   serverFiles:
     alerts:
       alertmanager.yml: |-
         global:
           resolve_timeout: 5m
         route:
           receiver: 'default'
         receivers:
         - name: 'default'
     alerting_rules.yml: |-
       groups:
         - name: example-alert-rules
           rules:
             - alert: HighCPUUsage
   ```

```
    expr:
        node_cpu_seconds_total{mode="idle"}
        < 0.2
    for: 5m
    labels:
      severity: critical
    annotations:
      summary: "Wysokie żzuycie CPU"
      description: "żUycie CPU
          łprzekroczyo próg 80% przez
          ostatnie 5 minut"
```

W tym przykładzie:

- `HighCPUUsage` to nazwa alertu.
- `expr` definiuje wyrażenie PromQL, które uruchamia alert, gdy czas procesora w trybie bezczynności jest mniejszy niż 20% przez 5 minut.
- `labels` i `annotations` dostarczają dodatkowych informacji o alertach.

3. **Zaktualizuj wdrożenie Prometheus za pomocą Helm**
   Po zapisaniu zmian w pliku `values.yaml`, zaktualizuj wdrożenie Prometheus w namespace `monitoring`, aby załadować nowe reguły alertów:

```
helm upgrade prometheus
    prometheus-community/prometheus -f
    values.yaml -n monitoring
```

4. **Sprawdź nowe reguły w interfejsie Prometheus**
   Przejdź do interfejsu Prometheus, zazwyczaj dostępnego na `/alerts` w UI Prometheus. Tam możesz zweryfikować, czy dodane reguły są widoczne w sekcji alertów.

Alert wysokiego zużycia CPU: Generuje alert, gdy zużycie CPU przekracza 80% przez więcej niż 5 minut.

```
- alert: HighCPUUsage
  expr:
      avg(rate(node_cpu_seconds_total{mode!="idle"}[5m]))
      > 0.8
  for: 5m
  labels:
    severity: critical
  annotations:
    summary: "High CPU usage detected"
    description: "The CPU usage is over 80% for the last
        5 minutes."
```

Aby wywołać ten alert, zmniejsz liczbę replik w Deployment `stress-ng` do zera, a następnie przywróć je, co wygeneruje gwałtowny wzrost zużycia CPU.
Alert wysokiego czasu odpowiedzi NGINX: Generuje alert, gdy średni czas odpowiedzi serwera NGINX przekracza 500 ms.

```
- alert: HighResponseTime
  expr: avg(nginx_http_response_time_seconds) > 0.5
  for: 5m
  labels:
    severity: warning
  annotations:
    summary: "High response time detected"
    description: "Average NGINX response time exceeds
       500 ms over 5 minutes."
```

Ten alert można wywołać, generując intensywny ruch do NGINX przy użyciu Apache Benchmark (AB), co zwiększy czas odpowiedzi.

6.5 Zadanie samodzielne

Skonfiguruj dodatkowy alert, który monitoruje zużycie pamięci na poziomie klastra. Twoje zapytanie powinno zwracać alert, gdy dostępna pamięć RAM na węzłach klastra spadnie poniżej 20%. Skorzystaj z interfejsu MailHog, aby zobaczyć powiadomienie w momencie wystąpienia zdarzenia.

7 Monitorowanie bazy danych PostgreSQL za pomocą PostgreSQL Exporter

W tym zadaniu skonfigurujesz bazę danych PostgreSQL w klastrze Kubernetes oraz kontener sidecar `PostgreSQL Exporter`, który będzie udostępniał metryki bazy danych na endpointzie `/metrics`. Dzięki temu będziesz mógł monitorować wydajność bazy danych w Prometheus i Grafana oraz skonfigurować alerty na podstawie wybranych metryk.

7.1 Cel

- Skonfigurować PostgreSQL w klastrze Kubernetes z `PostgreSQL Exporter` jako sidecar.
- Zbierać metryki dotyczące bazy danych PostgreSQL.
- Stworzyć dashboard w Grafana, który będzie wizualizował te metryki.
- Skonfigurować alert w Prometheus, który powiadomi o nadmiernym obciążeniu bazy danych.

7.2 Krok 1: Instalacja PostgreSQL z PostgreSQL Exporter jako sidecar

1. Stwórz plik YAML dla Deploymentu PostgreSQL z kontenerem sidecar `PostgreSQL Exporter`. Upewnij się, że kontener `postgresql` uruchamia bazę danych, a kontener `postgres-exporter` działa jako sidecar, łącząc się z bazą danych przy użyciu zmiennych środowisk wg dokumentacji. Do zbierania metryk wykorzystaj następujący obraz: `https://hub.docker.com/r/prometheuscommunity/postgres-exporter/`.

2. Zastosuj plik YAML, aby wdrożyć Deployment:

   ```
   kubectl apply -f postgresql-deployment.yaml
   ```

3. Utwórz Service dla PostgreSQL Exporter, aby `Prometheus` mógł zbierać metryki z `PostgreSQL Exporter`.

4. Zastosuj plik YAML dla Service:

   ```
   kubectl apply -f postgres-exporter-service.yaml
   ```

7.3 Krok 2: Konfiguracja Prometheus do monitorowania metryk PostgreSQL

1. Dodaj odpowiednie adnotacje do definicji YAML aby umożliwić autodiscovery przez Prometheusa.

2. Zaktualizuj konfigurację Prometheus i zrestartuj go, aby uwzględnić nową konfigurację.

7.4 Krok 3: Tworzenie dashboardu w Grafana

1. Skonfiguruj Grafana, aby wyświetlała metryki PostgreSQL, dodając nowe panele dla następujących metryk:

   - Liczba aktywnych połączeń do bazy danych PostgreSQL.
   - Użycie pamięci przez PostgreSQL.
   - Czas odpowiedzi zapytań SQL.
   - Liczba operacji odczytu i zapisu.

2. Eksperymentuj z różnymi typami wizualizacji, takimi jak wykresy liniowe, liczby (gauge) i histogramy, aby uzyskać pełniejszy obraz działania bazy danych.

7.5 Krok 4: Skonfigurowanie alertów w Prometheus

1. Dodaj regułę alertu dla liczby aktywnych połączeń, która będzie wywoływana, gdy liczba aktywnych połączeń przekroczy 80% maksymalnej liczby połączeń.

2. Zastosuj konfigurację alertu i upewnij się, że alert pojawi się w momencie przekroczenia wartości progowej.

7.6 Zadanie dodatkowe

Skonfiguruj dodatkowy alert, który monitoruje czas odpowiedzi zapytań SQL w PostgreSQL. Skonfiguruj alert, który zostanie wywołany, gdy średni czas odpowiedzi przekroczy 500 ms przez okres 5 minut. Sprawdź, czy alert pojawia się w MailHog, gdy obciążenie bazy danych jest wysokie.