ДОДАТОК А

Перелік джерел посилання за науковими напрямами керівника та науковців кафедри програмної інженерії

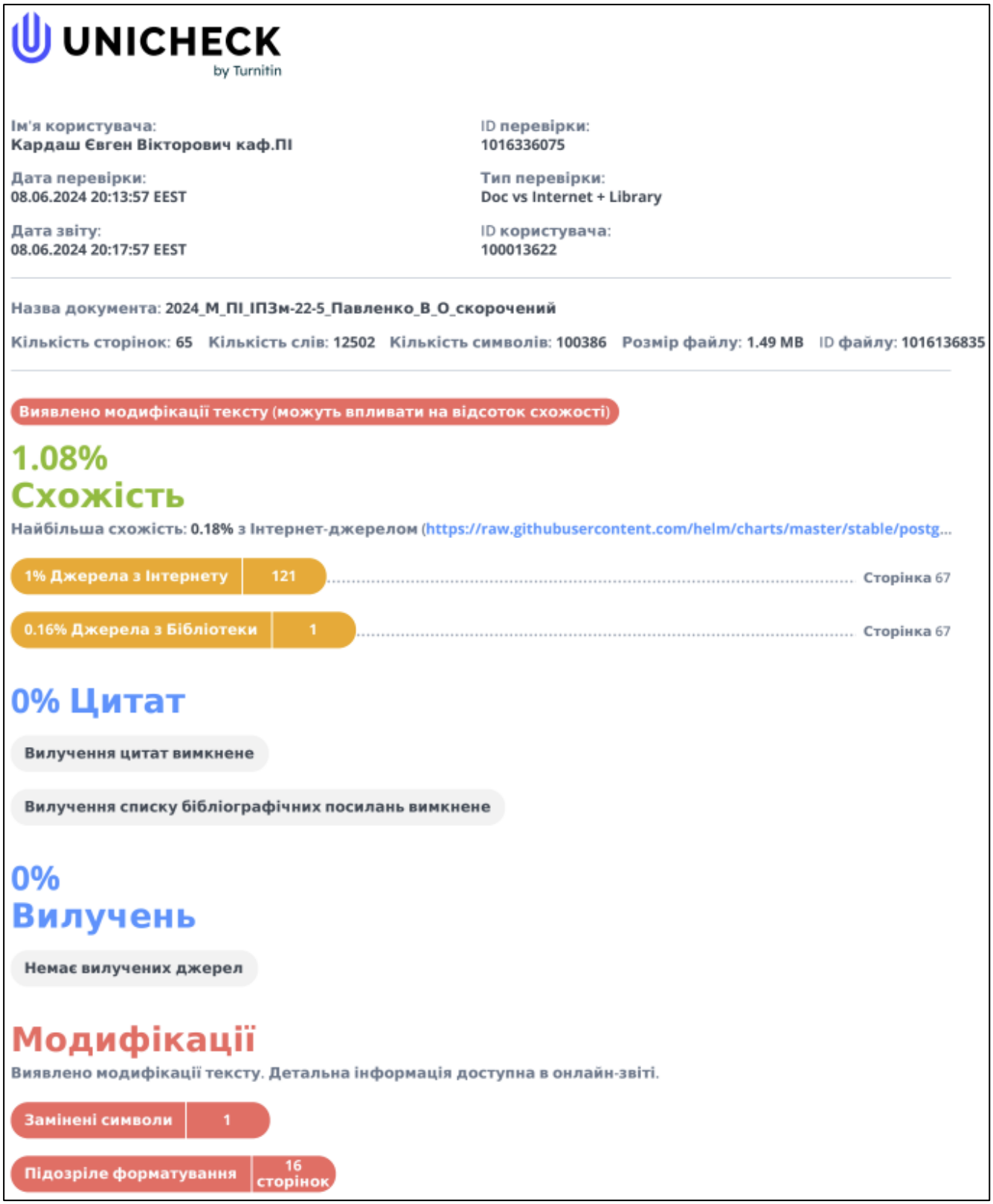
26. O. Makieiev, Study of methods of creating service-oriented software systems in Azure / N. Kravets Computer Systems and Information Technologies, 2023 (2), Pp. 38-47. Doi: 10.31891/csit-2023-2-5 (дата звернення 28.03.2024).

33. Шевченко Д.А., Evaluation and optimization of software product infrastructure, друк. "Журнал Wschodnioeuropejskie Czasopismo Naukowe (East European Scientific Journal), 58, ст.. 56-62, Printed in the ""Jerozolimskie 85/21, 02-001 Warsaw, Poland»" (дата звернення 30.03.2024).

64. Дудар З., Порівняння методів прогнозування часових рядів / Широкопетлєва М.С., Пономаренко О.А., 2018 Порівняння методів прогнозування часових рядів друк. Бионика интеллекта. Харьков.: ХНУРЭ, 2018. Вип.2 (91). С.41-47 (дата звернення 03.05.2024).

ДОДАТОК Б

Звіт результатів перевірки на унікальність тексту в базі ХНУРЕ



UNICHECK
by Turnitin

Ім'я користувача:
Кардаш Євген Вікторович каф.ПІ

ID перевірки:
1016336075

Дата перевірки:
08.06.2024 20:13:57 EEST

Тип перевірки:
Doc vs Internet + Library

Дата звіту:
08.06.2024 20:17:57 EEST

ID користувача:
100013622

Назва документа: 2024_М_ПІ_ІПЗм-22-5_Павленко_В_О_скорочений

Кількість сторінок: 65 Кількість слів: 12502 Кількість символів: 100386 Розмір файлу: 1.49 MB ID файлу: 1016136835

Виявлено модифікації тексту (можуть впливати на відсоток схожості)

1.08%
Схожість

Найбільша схожість: 0.18% з Інтернет-джерелом (https://raw.githubusercontent.com/helm/charts/master/stable/postg...

1% Джерела з Інтернету 121 ..... Сторінка 67

0.16% Джерела з Бібліотеки 1 ..... Сторінка 67

0% Цитат

Вилучення цитат вимкнене

Вилучення списку бібліографічних посилань вимкнене

0%
Вилучень

Немає вилучених джерел

Модифікації

Виявлено модифікації тексту. Детальна інформація доступна в онлайн-звіті.

Замінені символи 1

Підозріле форматування 16 сторінок

ДОДАТОК В

Слайди презентації

Харківський національний університет радіоелектроніки

Дослідження методів оркестрування та масштабування контейнеризованих систем у хмарних середовищах

Виконав Павленко В.О.
Керівник доц. Хацько Н.Є

Мета дослідження

Аналіз існуючих технологій та моделей оркестрування та масштабування розподілених хмарних інформаційних систем.

Актуальність дослідження

Швидкий розвиток технологій управління розподіленими системи.
Необхідність підвищення ефективності управління контейнеризованими додатками без збільшення складності налаштування інфраструктури.

Очікуваний результат дослідження

Створення універсального компоненту розгортання, який може бути перевикористаний для різних контейнеризованих додатків.
Реалізація компоненту автоматичного горизонтального масштабування на базі ресурсних та сервісних метрик з повною інтеграцію системи моніторингу Prometheus.

Контейнеризація додатків

Контейнеризація - це метод, який використовується для пакування програми разом з її залежностями та конфігураційними файлами в стандартизовану одиницю, яка називається контейнер.

Переваги контейнеризації:

- Ізоляція ресурсів та компонентів;
- Стандартизація середовища виконання;
- Забезпечення сумісності та консистентності між різними середовищами розробки та тестування.

Технологічні компоненти контейнеризації:

- **Рушій контейнерів**
  Створення, управління та виконання контейнерізованих додатків.
- **Простори імен операційної системи**
  Ізоляція імен PID, NET, MNT, IPC та користувачів.
- **Контрольні групи операційної системи**
  Ізоляція системних ресурсів (процесор, пам'ять).

Docker як стандарт індустрії в сфері контейнеризації

Docker є найпопулярнішою реалізацією рушія контейнерів та стандартом індустрії, надаючи повний набір інструментів і команд для створення, запуску та керування контейнерами.

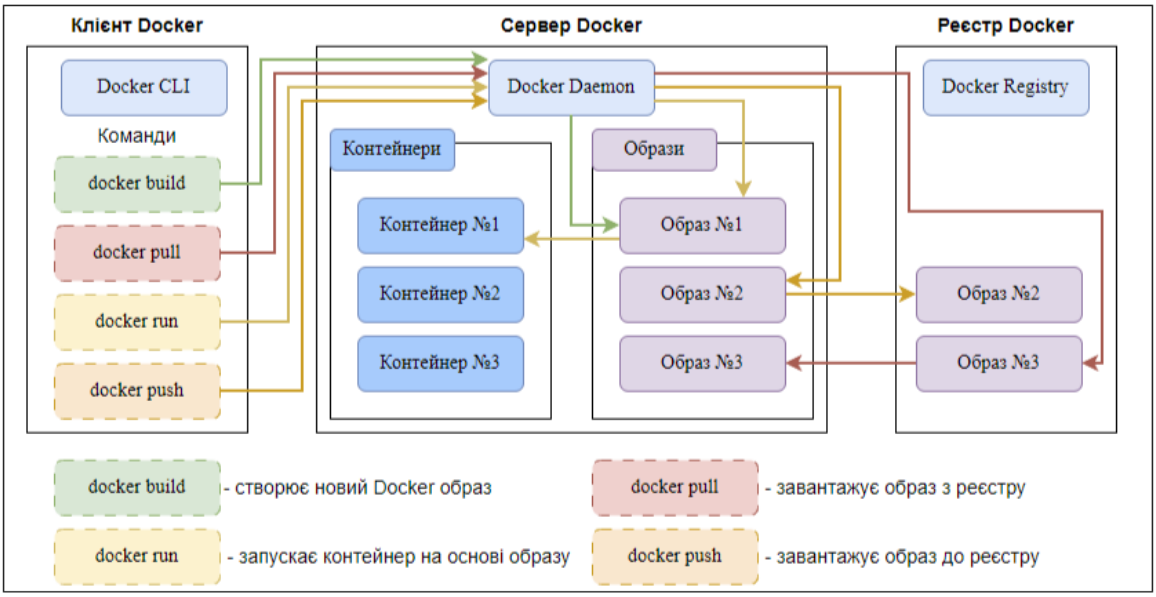
Компоненти загальної архітектури Docker:

- **Фоновий процес Docker Daemon**
  Відповідає за управління сутностями Docker – контейнерами, образами, мережами та томами.
- **Клієнт Docker CLI**
  Інтерфейс командного рядка, здійснює запити до Docker Daemon через REST API.
- **Docker Images**
  Образи контейнерізованих додатків у форматі Docker.
- **Docker Registry**
  Реєстр образів контейнеризованих додатків Docker.

Аналоги:

- Rocket
- Singularity

Взаємодія компонентів Docker на прикладі виконання команд Docker CLI



Створення Docker образу тестового додатку

Ключовим елементом у контейнеризації додатків у Docker є створення **Dockerfile**, який представляє собою текстовий файл, що містить набір інструкцій необхідних для створення образу контейнера.

Інструкції у Dockerfile – це ключові слова, які застосовують певну дію до образу, наприклад, копіювання файлу з робочого каталогу або виконання команди всередині образу.
Dockerfile підтримує 18 інструкцій для створення образів контейнеру.

Dockerfile для тестового додатку на Java

```
FROM amazoncorretto:21.0.1

WORKDIR /service
COPY build/libs/msjava-spring-prometheus-sample.jar msjava-spring-prometheus-sample.jar
CMD ["java", "-jar", "msjava-spring-prometheus-sample.jar"]
```

Оркестрування контейнеризованих додатків

Оркестрування контейнерів - процес автоматизації управління, розгортання, масштабування та моніторингу контейнеризованих додатків у великомасштабних розподілених системах.

Основні завдання оркестрування:

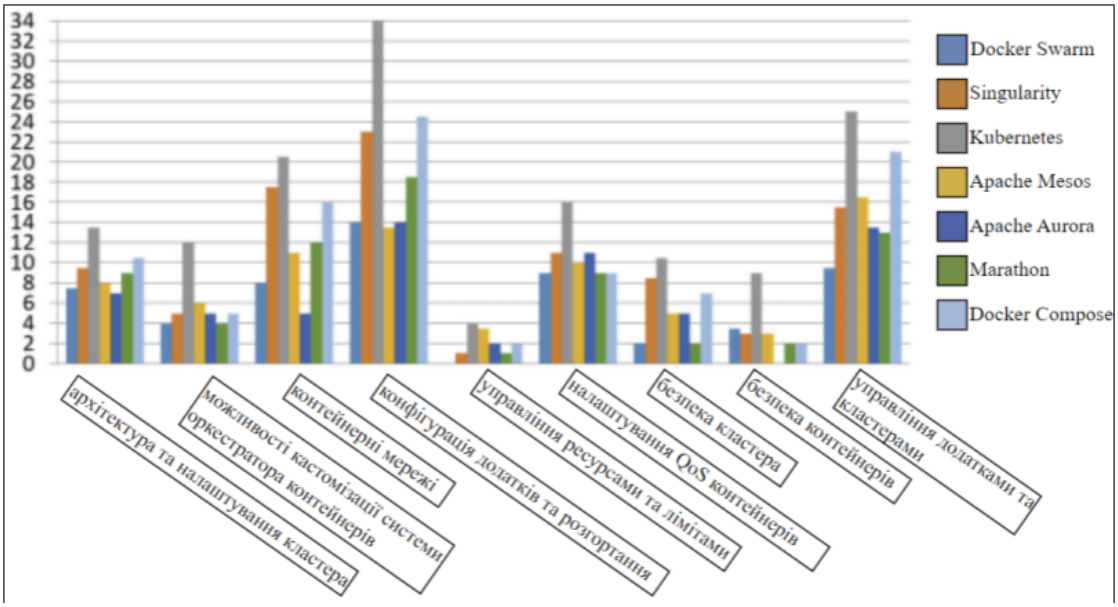
- Координація життєвого циклу контейнерів;
- Розподіл ресурсів між контейнерами;
- Забезпечення мережевої взаємодії та виявлення сервісів;
- Гарантування високої доступності і відмовостійкості.

Базові абстракції оркестрування:

- **Репліка**
  Найменша базова одиниця, яка представляє запакований та розгорнутий контейнер.
- **Група реплік**
  Абстракція, що керує кількістю та станом реплік одного розгорнутого додатку.
- **Вузол**
  Фізичний сервер або віртуальна машина, яка забезпечує ресурси для розгортання контейнерів.
- **Кластер**
  Набір вузлів, на яких виконуються контейнеризовані додатки.

Kubernetes (K8s) - найфункціональніший оркестратор

Дослідження «A Comprehensive Feature Comparison Study of Open-Source Container Orchestration Frameworks»



Маніфести — метод розгортання ресурсів в Kubernetes

Основним інструментом для опису і управління ресурсами в кластері Kubernetes є **маніфест**.

Маніфести описують декларативно у форматі YAML, вони служать інструкціями для Kubernetes про те, як розгортати і керувати компонентами системи, такими як контейнери, сервіси, мережеві політики тощо.

Маніфести визначають бажаний стан системи, а Kubernetes забезпечує його досягнення.

Основні ресурси, які можуть бути розгорнуті:

- Pod;
- ReplicaSet;
- Deployment;
- Service;
- ConfigMap.

Маніфест ресурсу Deployment для тестового додатку

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: diploma-sample-java-app-service
spec:
  replicas: 2
  selector:
    matchLabels:
      app: diploma-sample-java-app
  template:
    metadata:
      labels:
        app: diploma-sample-java-app
    spec:
      containers:
        - name: diploma-sample-java-app
          image: diploma-sample-java-app:latest
          resources:
            requests:
              memory: "256Mi"
              cpu: "250m"
          ports:
            - containerPort: 8080
```

Helm — альтернативний метод розгортання в K8s

Helm – шаблонізатор та менеджер пакетів, що спрощує процеси розгортання, оновлення та управління ресурсами в Kubernetes.

Переваги Helm над традиційним розгортанням:

- атомарність операцій;
- керування залежностями;
- спрощення управління;
- перевикористання маніфестів.

Пакет в **Helm** називається **Helm Chart** та складається з наступних елементів:

- **Chart.yaml** - метадані про чарт;
- **values.yaml** - значення за замовчуванням для параметрів в шаблонах;
- **templates/** - директорія з шаблонами маніфестів.

Шаблонізований ресурс Deployment

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{include "diploma-chart-sample.fullname" .}}
  labels:
    {{- include "diploma-chart-sample.labels" . | nindent 4}}
spec:
  replicas: {{.Values.replicaCount }}
  selector:
    matchLabels:
      {{- include "service.selectorLabels" . | nindent 6 }}
  template:
    spec:
      containers:
        - name: {{.Chart.Name}}
          image: "{{.Values.image.repository}}:{{.Values.image.tag}}"
          resources:
            {{- toYaml .Values.resources | nindent 12}}
          ports:
            - name: http
              containerPort: {{.Values.service.port}}
              protocol: TCP
```

Scheduler, механізм вибору вузла розгортання в K8s

Фаза відсіювання вузлів

- **Перевірка кількості вільних ресурсів**
- **Механізм Node Affinity/Anti-Affinity**
  Розгортання реплік на вузлах, що мають певний набір характеристик.
- **Механізм Pod Affinity/Anti-Affinity**
  Розміщення реплік на вузлах відносно інших розгорнутих реплік.
  Наприклад, всі репліки додатку мають бути розгорнуті на одному вузлі або на різних вузлах.
- **Taints та Tolerations**
  Taints запобігають розміщенню реплік на вузлах без відповідних Tolerations.

Фаза ранжування вузлів

- **Resource Allocation Scoring**
  Загальна оцінка кількості вільних ресурсів доступних для розгортання на вузлі.

$$Score = \frac{Request_{resource}}{Capacity_{resource}}$$

- **Balanced Resource Allocation Scoring**
  Загальне співвідношення використання процесору та пам'яті.

$$Score = 1 - \left| \frac{\frac{Usage_{cpu}}{Capacity_{cpu}} - \frac{Usage_{memory}}{Capacity_{memory}}}{\frac{Usage_{cpu}}{Capacity_{cpu}} + \frac{Usage_{memory}}{Capacity_{memory}}} \right|$$

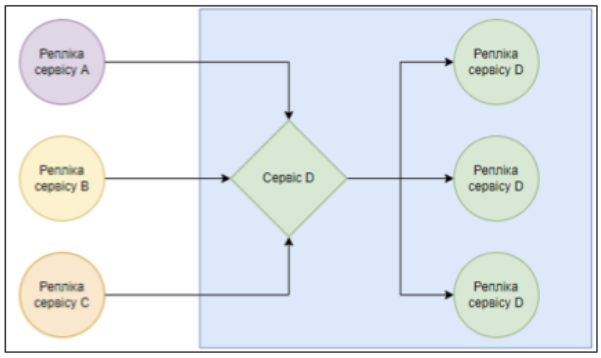
Service Discovery та балансування навантаження

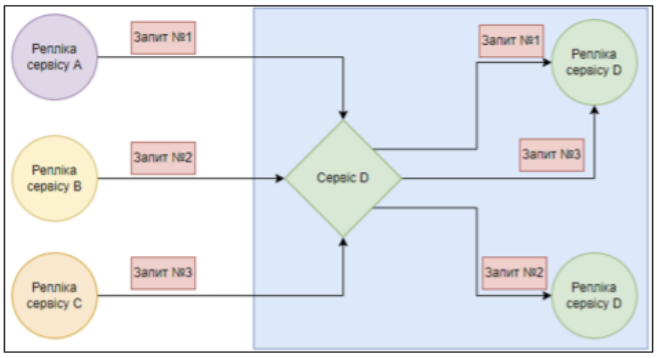
Service Discovery механізм забезпечується Kubernetes ресурсом **Service**.

В ньому визначаються правила та кінцеві точки доступу до групи реплік.

Сервісам-клієнтам не треба знати інформацію про порти кожної репліки, достатньо внутрішньої IP-адреси абстракції Service.

Ресурс **Service** з типом **LoadBalancer** забезпечує балансування навантаження між репліками алгоритмом **Round Robin**.





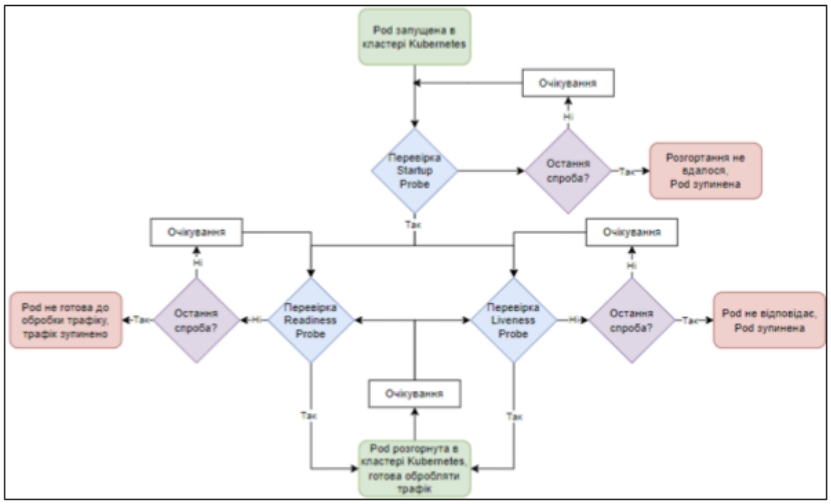
Механізм Health Check в оркестраторі Kubernetes

Механізм **Health Check** (перевірка працездатності) використовується для визначення статусу розгорнутого контейнеризованого додатку в кластері Kubernetes.

Конфігурація Health Check

```yaml
livenessProbe:
  enabled: true
  initialDelaySeconds: 30
  periodSeconds: 10
  timeoutSeconds: 5
  failureThreshold: 6
  successThreshold: 1
  httpGet:
    path: /actuator/health
    port: 8080

readinessProbe:
  enabled: true
  initialDelaySeconds: 30
  periodSeconds: 10
  timeoutSeconds: 5
  failureThreshold: 6
  successThreshold: 1
  httpGet:
    path: /actuator/health
    port: 8080
```

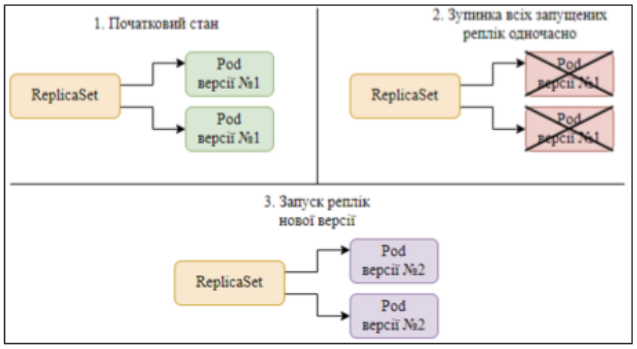
Алгоритм Health Check



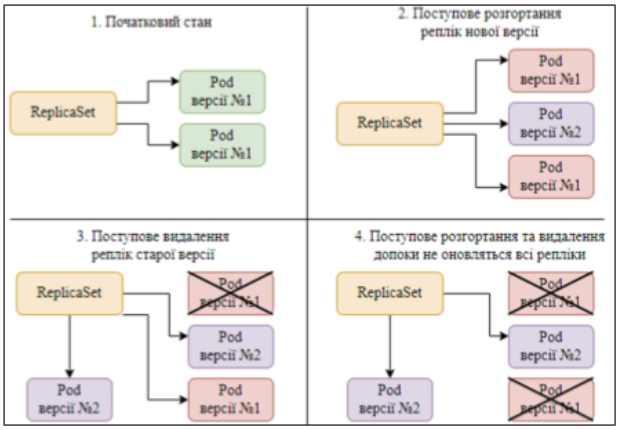
Стратегії оновлення реплік інтегровані в Kubernetes

Стратегії оновлення реплік визначають методи управління процесом оновлення контейнерів у кластері, тобто як і коли репліки будуть замінені на нові версії.

Стратегія Recreate



Стратегія Rolling Update



Автоматичне масштабування реплік в Kubernetes

Автоматичне горизонтальне масштабування в Kubernetes відбувається за рахунок вбудованого компоненту **Horizontal Pod Autoscaler (HPA)**.

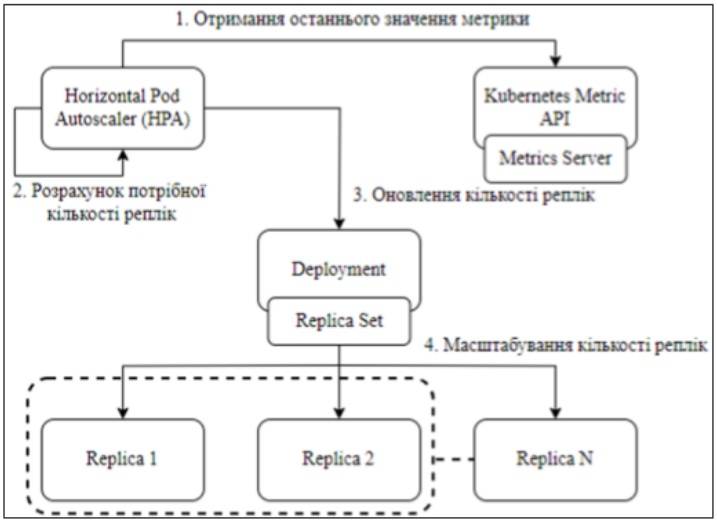
HPA розгортається в кластері за принципом «**один Deployment – один екземпляр компоненту HPA**», тож кожна група реплік має відокремлену розгорнуту сутність **HPA** зі специфічною конфігурацією.

Формула для розрахунку кількості реплік додатку

$$R = \frac{M_c}{M_t}$$

$$P_t = P_c \times R$$

Алгоритм роботи HPA



Традиційний метод масштабування: HPA + Metrics Server

Метод з використанням **HPA + Metrics Server** передбачає автоматичне масштабування на базі **ресурсних метрик** розгорнутих реплік: утилізація процесору та пам'яті.

Kubernetes Metrics Server – це сервіс розроблений Kubernetes Community, який збирає **метрики використання ресурсів з Kubelet-ів** та надає до них доступ через **Kubernetes Metric API**.

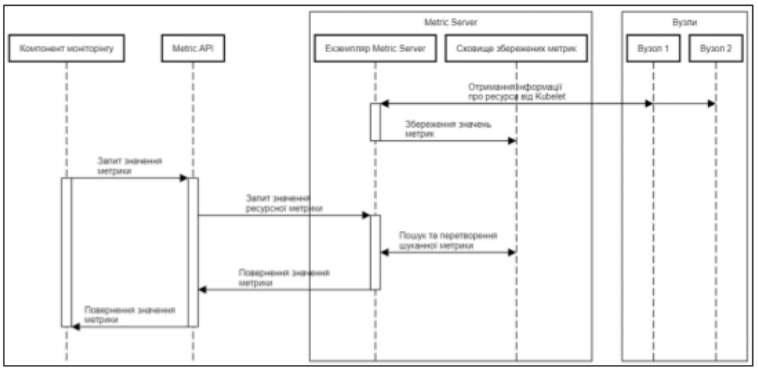
Переваги методу:

- простота розгортання інфраструктури;
- автоматичний збір значень метрик;
- масштабування на базі ресурсних метрик гарантує, що жодна репліка не перетне ліміт використання ресурсів.

Недоліки методу:

- масштабування на базі ресурсних метрик не відображає реальне навантаження на систему та якість обслуговування користувачів.

Діаграма послідовності взаємодії Metric API з Metrics Server



Традиційний метод масштабування: HPA + Prometheus

Prometheus є стандартом індустрії у сфері **моніторингу**, **збирання сервісних та ресурсних метрик**, створений для роботи у високодинамічних контейнерних середовищах.

Інтеграція розгорнутої інфраструктури **Prometheus** з **Kubernetes HPA** дозволяє реалізовувати більш складні сценарії масштабування кількості реплік, використовуючи не лише ресурсні метрики, але й сервісні, які збирає Prometheus.

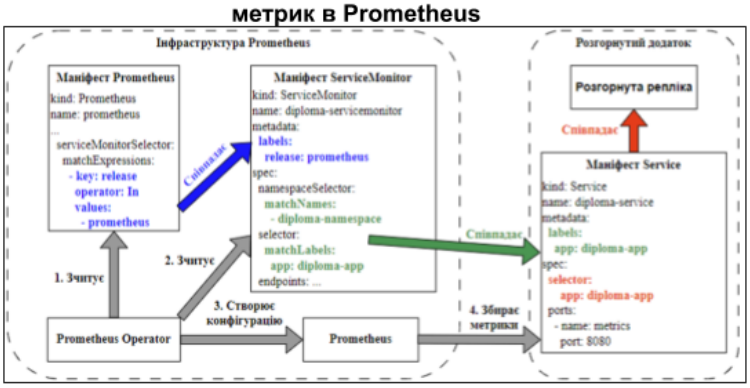
Переваги методу:

- масштабування на базі будь-яких метрик: ресурсних та сервісних.
- мова запитів PromQL дозволяє виконувати складні маніпуляції з часовими рядами Prometheus та використовувати їх у якості цільової метрики.

Недоліки методу:

- складність налаштування інфраструктури Prometheus у кластері;
- необхідність адаптування метрик за допомогою Prometheus Adapter для використання їх в Kubernetes HPA.

Алгоритм пошуку та створення конфігурації для збору метрик в Prometheus



Розробка компоненту масштабування та універсальної моделі розгортання додатків в Kubernetes

Мета розробки

Створення компоненту автоматичного масштабування для оркестратора Kubernetes на базі ресурсних та сервісних метрик Prometheus, та створення універсального шаблонізованого методу розгортання додатків з підтримкою реалізованого компоненту за замовчуванням.

Причини розробки

- Відсутність у загальному доступі універсальних, шаблонізованих моделей для розгортання контейнеризованих додатків, що інтегрують базові функції Kubernetes.
- Обмеження використання запитів PromQL напряму в компоненті масштабування як цільової метрики.
- Потреба у додатковому розгортанні та конфігуруванні контейнеризованих застосунків та ресурсів Kubernetes для налаштування автоматизованого масштабування на базі ресурсних та сервісних метрик у традиційних методах – HPA, Metrics Server, Prometheus Adapter тощо.

Постановка задачі

- підтримка індивідуального налаштування параметрів масштабування для кожної групи реплік;
- використання PromQL запиту як цільової метрики без необхідності адаптування;
- механізм стабілізації групи реплік після проведення операцій масштабування;
- створення універсального шаблону розгортання реплік на базі Helm чарт з автоматичною підтримкою розробленого компоненту масштабування.

Архітектура розробленого компоненту масштабування

Компонент реалізовано на Java 21 з використанням фреймворку Spring Boot 3 та офіційної бібліотеки Kubernetes для доступу до Kubernetes API.

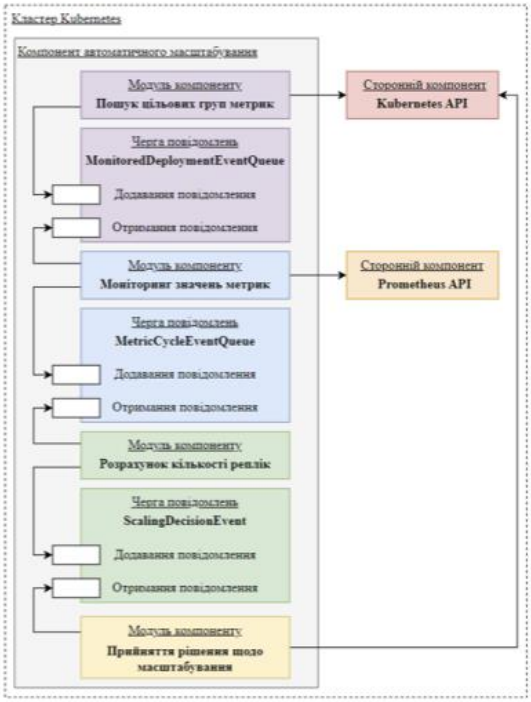
Модулі компоненту:

- модуль пошуку цільових груп реплік та їхньої конфігурації масштабування;
- модуль моніторингу значень метрик Prometheus;
- модуль розрахунку необхідної кількості реплік;
- модуль прийняття рішень щодо масштабування.

Взаємодія між модулями побудована за допомогою патерну Message Queue на базі інтерфейсу BlockingQueue з Java SDK.

Черги повідомлень розробленого компоненту:

- MonitoredDeploymentEventQueue;
- MetricCycleEventQueue;
- ScalingDecisionEvent.



Реалізація індивідуального конфігурування параметрів масштабування груп реплік

Для реалізації індивідуального конфігурування параметрів масштабування в універсальний шаблон розгортання було додано шаблон ресурсу ConfigMap та розширено шаблон ресурсу Deployment.

Розширення шаблону Deployment міткою цільової групи реплік для компоненту масштабування

```
{{- if and .Values.autoscaling.enabled}}
annotations:
  prometheus-autoscaler-target: "true"
{{end}}
```

Таким чином, для додатків з увімкненою функцію автоматичного масштабування («autoscaling.enabled: true»), під час розгортання буде автоматично створено ресурс ConfigMap, в якому будуть заповнені значення для всіх передбачених конфігураційних параметрів, а сутність Deployment буде проанотована як ціль масштабування.

Шаблон ресурсу ConfigMap з для конфігурації масштабування

```
{{- if and (.Values.autoscaling.enabled) (eq .Values.autoscaling.mode "PROMETHEUS_AUTOSCALER") }}
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ include "service.fullname" . }}-prometheus-autoscaling-config
  labels:
    {{- include "service.selectorLabels" . | nindent 4 }}
data:
  min-pods: "{{ .Values.autoscaling.minReplicas }}"
  max-pods: "{{ .Values.autoscaling.maxReplicas }}"
  prometheus-query: "{{ .Values.autoscaling.prometheusAutoscaler.prometheusQuery }}"
  threshold: "{{ .Values.autoscaling.prometheusAutoscaler.threshold }}"
  metric-window-seconds: "{{ .Values.autoscaling.prometheusAutoscaler.policy.metricWindowSeconds }}"
  cooldown-seconds: "{{ .Values.autoscaling.prometheusAutoscaler.policy.cooldownSeconds }}"
{{end}}
```

Реалізація модулю пошуку цільових груп реплік та їхньої конфігурації

Пошук цільових груп реплік для компоненту масштабування реалізовано за допомогою постійного сканування кластеру на наявність ресурсів Deployment з відповідною міткою-анотацією та ConfigMap, які зберігають інформацію про конфігурацію масштабування групи.

Після виявлення, оновлення або видалення цілей масштабування, модуль надсилає повідомлення в чергу MonitoredDeploymentEventQueue з вказанням відповідного типу повідомлення.

Реалізація сканування кластеру Kubernetes

```
@Scheduled(fixedDelayString = "${monitoring.target-scanner.rate}")
void scanMonitoredDeployments() throws ApiException {
    var deployments = appsClient.listNamespacedDeployment(monitoringConfiguration.getNamespace()).execute()
            .getItems().stream()
            .filter(this::isTargetDeployment)
            .toList();
    var configMaps = coreClient.listNamespacedConfigMap(monitoringConfiguration.getNamespace()).execute()
            .getItems().stream()
            .filter(this::isTargetDeploymentAutoscalingConfiguration)
            .toList();
    var linkedConfiguration = linkDeploymentToConfiguration(deployments, configMaps);
    var composedConfigurations = new LinkedList<MonitoredDeploymentConfiguration>();
    for (var linkedConfigurationEntry : linkedConfiguration.entrySet()) {
        try {
            var composedConfiguration = monitoredDeploymentComposer.composeMonitoredDeploymentConfiguration(
                    linkedConfigurationEntry.getKey(), linkedConfigurationEntry.getValue());
            composedConfigurations.add(composedConfiguration);
        } catch (MonitoredDeploymentComposingException composingException) {
            log.warn("Failed to compose monitored deployment configuration", composingException);
        }
    }
    log.info("Found [{}] monitored deployment targets, configuration [{}]",
            composedConfigurations.size(),
            composedConfigurations);
    monitoredDeploymentService.updateMonitoredDeployments(composedConfigurations);
}
```

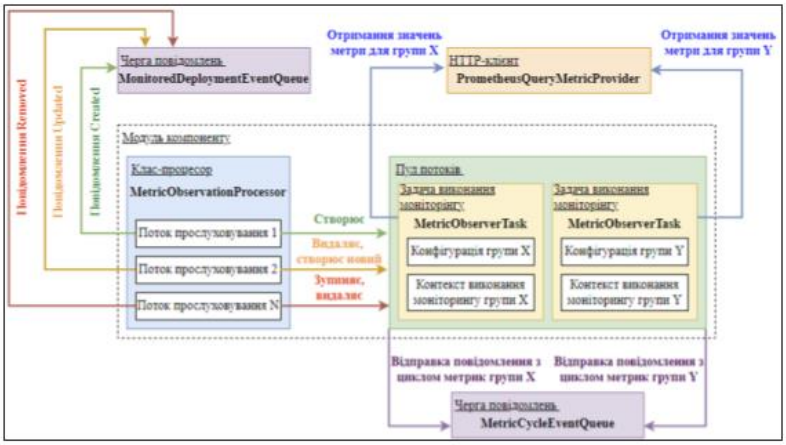
Реалізація модулю моніторингу цільових метрик

Модуль прослуховує повідомлення про знайдені цільові групи реплік для масштабування та створює для кожної групи асинхронну задачу обстеження значень метрики.

Інноваційністю реалізації та відмінністю від традиційних методів є збирання **циклів метрик** на базі **запитів PromQL без необхідності їхнього адаптування за допомогою Prometheus Adapter**, що спрощує процес налаштування масштабування для нових додатків та значно знижує вплив аномалій та випадкових коливань значень метрик.

Цикл представляє собою серію метричних даних, зібраних протягом фіксованих інтервалів часу.

Архітектура модулю моніторингу



Реалізація модулю розрахунку кількості реплік

Модуль прослуховує повідомлення з черги MetricCycleEventQueue, тобто отримує останній цикл зібраних метрик для групи реплік та базі циклу розраховує необхідну кількість реплік для дійсного навантаження.
У випадку, якщо розрахована кількість реплік не збігається з дійсною, модуль відправляє повідомлення до черги ScalingDecisionEventQueue.

Формула для розрахунку необхідної кількості реплік

$$P_t = \min\left(\max\left(\left\lceil P_c \times \frac{M_a}{M_t} \right\rceil, P_{min}\right), P_{max}\right)$$

де M_a – середнє арифметичне значення метрики для останнього циклу,

M_t – порогове значення метрики,

P_{min} – мінімальна кількість реплік,

P_{max} – максимальна кількість реплік,

P_c – дійсна кількість реплік,

P_t – цільова кількість реплік.

Реалізація розрахунку необхідної кількості реплік

```
private void processMetricCycleEvent(MetricCycleEvent metricCycleEvent) throws ApiException {
    var target = metricCycleEvent.getTarget();
    var namespace = monitoringConfiguration.getNamespace();

    var deploymentScale = kubernetesAppsClient.readNamespacedDeploymentScale(target.getName(), namespace)
            .execute();
    if (deploymentScale == null) {
        log.info("Deployment scale [{}] doesn't exist", target.getName());
        return;
    }
    var currentReplicasCount = Optional.ofNullable(deploymentScale.getSpec()) Optional<V1ScaleSpec>
            .map(V1ScaleSpec::getReplicas) Optional<Integer>
            .orElseThrow(() -> new NullPointerException("Replicas count is null, processing is unavailable"));
    var metricRatio = calculateMetricToThresholdRatio(
            metricCycleEvent.getLastObservationCycle(), target.getPrometheusQueringConfiguration().getThreshold());
    var desiredReplicasCount = Math.ceil(currentReplicasCount * metricRatio);
    var updatedReplicaCount = (int) Math.min(
            Math.max(desiredReplicasCount, target.getLimitConfiguration().getMinimumPods()),
            target.getLimitConfiguration().getMaximumPods());

    if (updatedReplicaCount != currentReplicasCount) {
        var scalingDecisionEvent = new ScalingDecisionEvent(currentReplicasCount, updatedReplicaCount, target);
        scalingDecisionEventSubmitter.submitEvent(scalingDecisionEvent);
        log.info("Decided new replica count for deployment [{}] - [{} replicas]",
                target.getName(), updatedReplicaCount);
    } else {
        log.info("Decided replicas count is equal to current replicas count for deployment [{}], applying ignored",
                target.getName());
    }
}

private double calculateMetricToThresholdRatio(List<MetricObservation> lastCycleObservations, double threshold) {
    var avgMetricValueForLastCycle = lastCycleObservations.stream() Stream<MetricObservation>
            .map(MetricObservation::getMetricValue) Stream<Double>
            .mapToDouble(Double::doubleValue) DoubleStream
            .average() OptionalDouble
            .orElse( other: 0);
    return avgMetricValueForLastCycle / threshold;
}
```

Реалізація модулю прийняття рішень масштабування

Реалізований механізм прийняття рішення базується на концепції **Cooldown Period** (період стабілізації). Цей механізм запобігає надмірному або занадто частому масштабуванні кількості реплік, що призводить до нестабільної роботи системи та неефективному використанню ресурсів.

Після кожної операції масштабування цільовій групі реплік потрібен час для стабілізації, це дозволяє їй адаптуватися до нової кількості розгорнутих ресурсів, тож період стабілізації є корисним у випадках, коли значення метрики можуть швидко змінюватися від суттєвих коливань навантаження.

Реалізація механізму Cooldown Period для прийняття рішення щодо масштабування

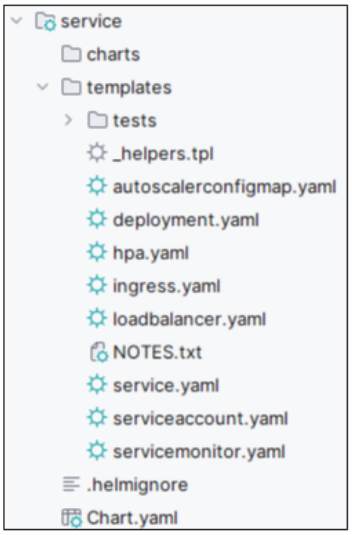
```
private void processScalingDecisionEvent(ScalingDecisionEvent event) throws ApiException {
    var previousAppliedScalingDecision = appliedScalingDecisions.get(event.getTarget().getName());
    if (previousAppliedScalingDecision != null
            && !isAfterCooldownPeriod(previousAppliedScalingDecision, event.getTarget())) {
        log.info("Scaling decision applying skipped for deployment [{}] due to cooldown period",
                event.getTarget().getName());
        return;
    }
    updateDeploymentReplicasCount(event.getDesiredReplicasCount(), event.getTarget());
    var appliedScalingDecision = new AppliedScalingDecision(
            Instant.now(), event.getCurrentReplicasCount(), event.getDesiredReplicasCount());
    appliedScalingDecisions.put(event.getTarget().getName(), appliedScalingDecision);
    log.info("Applied new replica count for deployment [{}] is [{}]",
            event.getTarget().getName(), event.getDesiredReplicasCount());
}

private boolean isAfterCooldownPeriod(
        AppliedScalingDecision previousAppliedScalingDecision,
        MonitoredDeploymentConfiguration target) {
    var now = Instant.now();
    var lastScalingOperationTimestamp = previousAppliedScalingDecision.getTimestamp();
    var afterCooldownPeriodTimestamp = lastScalingOperationTimestamp.plusSeconds(
            target.getCooldownConfiguration().getAfterScalingOperationCooldownSeconds());
    return now.isAfter(afterCooldownPeriodTimestamp);
}

private void updateDeploymentReplicasCount(
        int newReplicasCount,
        MonitoredDeploymentConfiguration target) throws ApiException {
    var patchBody = getPatchReplicasBody(newReplicasCount);
    PatchUtils.patch(V1Scale.class,
            () -> kubernetesAppsClient.patchNamespacedDeploymentScale(
                    target.getName(), monitoringConfiguration.getNamespace(),
                    new V1Patch(patchBody)
            ).buildCall( callback: null),
            V1Patch.PATCH_FORMAT_JSON_MERGE_PATCH,
            kubernetesAppsClient.getApiClient());
}
```

Реалізація універсального шаблону розгортання

До розробленого універсального шаблону увійшли маніфести наступних ресурсів Kubernetes:

- **Маніфест ресурсу Deployment.**
  Контролює життєвий цикл групи реплік, та надає Kubernetes налаштування механізму Health Check, стратегії оновлення та помічається як цільовий ресурс для компоненту автоматичного масштабування у разі, якщо для додатку, який розгортається, функція увімкнена;
- **Маніфест ресурсу ServiceMonitor.**
  Надає інформацію системі Prometheus про порти та кінцеві точки, з яких мають збиратися метрики;
- **Маніфест ресурсу LoadBalancer.**
  Конфігурує балансування запитів між репліками групи та надає порти для доступу до них ззовні та зсередини кластера;
- **Маніфест ресурсу ConfigMap.**
  Використовується як шаблон конфігурації для параметрів автоматичного масштабування групи реплік, які будуть використані розробленим компонентом.

Структура розробленого універсального шаблону для розгортання в Kubernetes на базі Helm Chart



Експериментальне дослідження методів масштабування

Мета дослідження

Визначення сильних сторін традиційних методів автоматичного масштабування та розробленого компоненту у різних експлуатаційних умовах методом порівняння якісних характеристик на базі отриманих даних.

Додаткова мета дослідження

Пошук слабких сторін для подальшої оптимізації розробленого компоненту автоматичного масштабування та універсального шаблону розгортання у порівняння з традиційними методами.

Постановка експерименту

- Мінімальна кількість реплік – 4;
- Максимальна кількість реплік - 24;
- Масштабування здійснюється на базі ресурсної метрики – середнє використання центрального процесору групою реплік;
- Порогове значення метрики - 90%;
- Період стабілізації для всіх методів - 90 секунд.

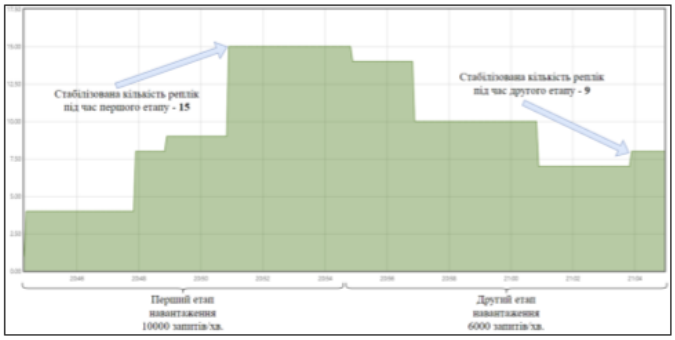
Додаток, який буде масштабуватися реалізовано так, що під час отримання HTTP-запиту, він інтенсивно використовує центральний процесор.

Навантаження подається на групу реплік в 2 етапи:

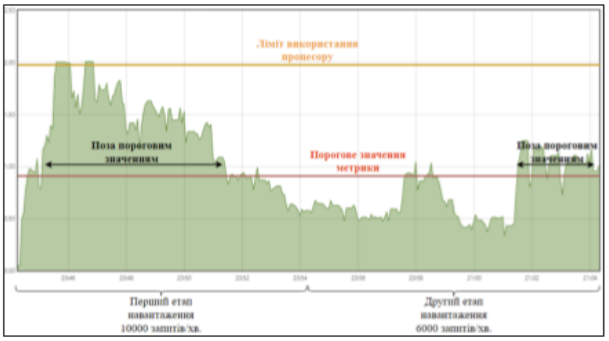
- 10000 запитів на хвилину протягом 10 хвилин;
- 6000 запитів на хвилину протягом 10 хвилин.

Результати дослідження: метод HPA + Metrics Server

Стан групи реплік під час проведення експерименту

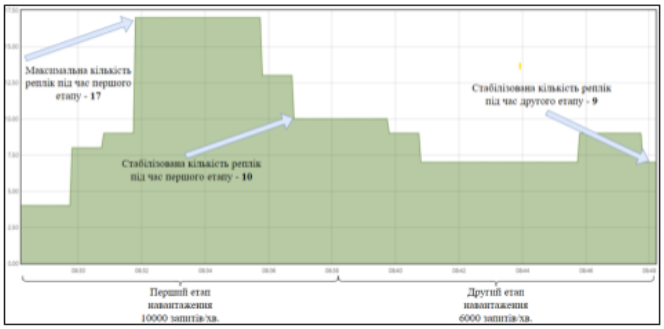


Середнє значення цільової метрики під час проведення експерименту



Результати дослідження: метод HPA + Prometheus

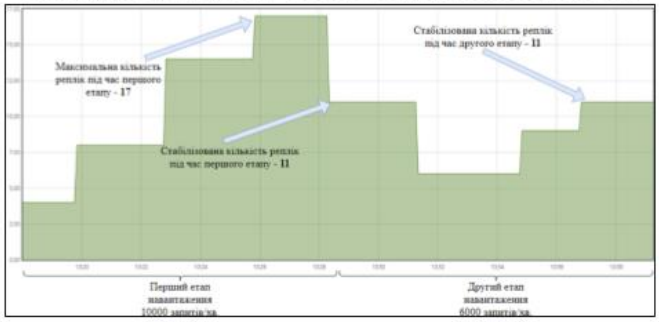
Стан групи реплік під час проведення експерименту



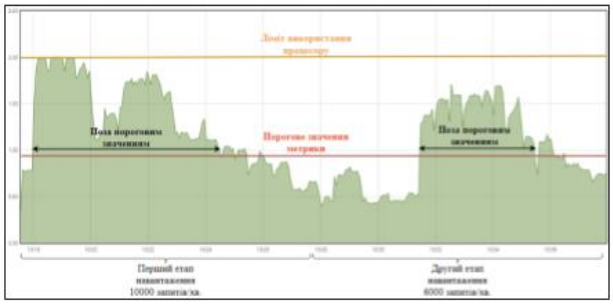
Середнє значення цільової метрики під час проведення експерименту

Результати дослідження: розроблений компонент

Стан групи реплік під час проведення експерименту



Середнє значення цільової метрики під час проведення експерименту



Загальні результати дослідження

Критерій порівняння	HPA, Metrics Server	HPA, Prometheus, Prometheus Adapter	Розроблений компонент
Перший етап навантаження (різке високе навантаження)			
Кількість операцій для повної стабілізації	3	5	4
Максимальна кількість реплік	15	17	17
Стабілізована кількість реплік	15	10	11
Час поза пороговим значенням (секунд)	425 с.	297 с.	306 с.
Час в зоні надвисокого навантаження	63 с.	20 с.	49 с.

Критерій порівняння	HPA, Metrics Server	HPA, Prometheus, Prometheus Adapter	Розроблений компонент
Другий етап навантаження (зниження навантаження після високого)			
Кількість операцій для повної стабілізації	4	3	3
Максимальна кількість реплік	14	9	11
Стабілізована кількість реплік	8	9	11
Час поза пороговим значенням (секунд)	211 с.	147 с.	212 с.

Висновки

Детально досліджені механізми контейнеризації на базі Docker та внутрішньої роботи оркестратора Kubernetes, такий як вибір обчислювального вузла для розгортання реплік, та найкращі практики оркестрування контейнеризованих додатків – постійна перевірка стану реплік, динамічна маршрутизація запитів та стратегії оновлення.

Розглянуто два традиційних концептуально різних метода масштабування реплік в Kubernetes. Знайдено переваги та недоліки кожного методу.
Об'єднання найкращих аспектів обох методів послугувало основою для створення власного компоненту автоматичного масштабування для Kubernetes.

Виведено власну концепцію збору метрик для масштабування та інтегровано її в розробленому компоненті. Збирання циклів метрик посприяло значному зниженню впливу аномалій та коливань значень метрик на розрахунок необхідної кількості реплік.

Традиційні методи та розроблений компонент було досліджено та порівняно на практиці в рівних умовах для високого та середнього навантаження. Отримані результати дослідження вказують, що є місце оптимізації розробленого компоненту, проте в дійсному стані він є конкурентоспроможним та може застосовуватися не тільки в контрольованих експериментальних умовах, а й у реальних проєктах з високими вимогами щодо адаптування стану ресурсів системи до змін навантаження.

Дякую за увагу!

ДОДАТОК Г

Апробація результатів роботи

Публікація тез доповіді для Міжнародної науково-практичної конференції «Інформаційні технології: наука, техніка, технологія, освіта, здоров'я. MicroCAD» 2024, Харків.

Інформаційні технології: наука, техніка, технологія, освіта, здоров'я. MicroCAD-2024

РОЗРОБКА КОМПОНЕНТУ АВТОМАТИЧНОГО ГОРИЗОНТАЛЬНОГО МАСШТАБУВАННЯ РЕПЛІК В KUBERNETES НА ОСНОВІ МЕТРИК PROMETHEUS ТА ШАБЛОНУВАННЯ HELM

Павленко В.О., Хацько Н.Є.

Харківський національний університет радіоелектроніки, м. Харків

Традиційний метод горизонтального масштабування в Kubernetes є малоефективним для великих розподілених систем [1-2], тому що він базується лише на основі показників утилізації системних ресурсів (ЦП, пам'ять, мережа) та не враховує дійсні параметри швидкодії додатків - кількість необроблених повідомлень в черзі, час на обробку запитів користувачів тощо.

Для вирішення цієї проблеми розроблено компонент, який дозволяє масштабувати кількість реплік в Kubernetes на основі користувацьких метрик з Prometheus (будь-яких метрик, які збирає Prometheus з розгорнутих додатків). Розроблений компонент складається з двох частин: шаблон Helm для розгортання додатку та компонент автоматичного масштабування.

Шаблон Helm робить додаток масштабованим, автоматично конфігуруючи сутності Deployment та ConfigMap з конфігурацією масштабування відповідного додатку під час розгортання в Kubernetes.

Розроблений компонент автоматичного масштабування розгортається в кластері Kubernetes за принципом «один кластер – одна репліка компоненту» та відповідає за масштабування додатків лише в своєму кластері. Компонент складається з трьох логічних частин, які обмінюються повідомленнями:

1. Пошук та ідентифікація додатків, які мають бути масштабованими;
2. Сканування Prometheus метрики ідентифікованих для масштабування додатків за принципом «один додаток – один паралельний процес»;
3. Розрахунок необхідної кількості реплік відповідно до останнього циклу сканування значення метрики за наступною формулою та прийняття рішення щодо масштабування:

$$R = \frac{M_c}{M_t},\ P_t = P_c \times R,$$

де M_c – дійсний показник метрики,
M_t – цільовий показник метрики,
R – співвідношення дійсного показника метрики до цільового (ratio),
P_c – дійсна кількість розгорнутих реплік,
P_t – цільова кількість розгорнутих реплік.

Отже, розроблений компонент та шаблон Helm може повністю замінити та розширити традиційний метод масштабування Kubernetes, дозволяючи масштабувати додатки на базі будь-яких метрик зібраних Prometheus.

Література:

1 Nikulina O. M., Khatsko K.O. Method of converting the monolithic architecture of a front-end application to microfrontends. *Вісник НТУ «ХПІ»*, 2023. – № 2 (10). – С. 79–84. DOI: 10.20998/2079-0023.2023.02.12

2 Zamkovyi M., Gavrylenko S., Khatsko K., Khatsko N. "Algorithmic Support for Building a Distributed IoT System in a Cloud Service, *IEEE 4th KhPI Week on Advanced Technology* 2023. pp. 1-6, DOI: 10.1109/KhPIWeek61412.2023.10312994.

ДОДАТОК Д

Експертний висновок результатів перевірки кваліфікаційної роботи на відповідність оформлення вимогам ДСТУ 3008: 2015

Експертний висновок результатів перевірки кваліфікаційної роботи

студент (посада) | програмної інженерії (кафедра) | ІПЗм-22-5 (група)

Павленко Владислав Олександрович
(прізвище, ім'я, по батькові)

Зауваження

Пункт ДСТУ 3008-2015	Зміст пункту	Сторінка кваліфікаційної роботи
1	2	3
	7.1 Загальні положення	
	7.3 Нумерація сторінок звіту	
	7.4 Нумерація розділів, підрозділів, пунктів, підпунктів	
	7.5 Рисунки	
	7.6 Таблиці	
	7.7 Переліки	
	7.8 Примітки	
	7.9 Виноски	
	7.10 Формули та рівняння	
	7.11 Посилання	
	7.13 Список авторів	
	7.14 Скорочення та умовні познаки	
	7.15 Додатки	

зауважень немає

Експерт _____ (підпис) Олена ОЛІЙНИК (прізвище, ініціали)

11.06.2024