Підвищення відмовостійкості (хмарного безсерверного) мікросервісного програмного забезпечення шляхом автоматизованого функціонального та відмовостійкого тестування.

Кузьмич О.М., Сенів М.М.

Анотація

Ключові слова (5-7 слів).

Вступ

Актуальність проблеми відмовостійкості мікросервісного ПЗ.

Огляд існуючих методів та їх обмеження.

Мета та завдання дослідження.

Основна частина

Огляд літератури

Аналіз останніх досліджень з теми відмовостійкості мікросервісів.

Класифікація відмов та методів їх обробки.

Опис концепції "пентагону тестування"

Розподілений характер мікросервісної архітектури та велика кількість потенційних точок відмови значно ускладнюють забезпечення безперебійної роботи програмного забезпечення. Відомо, що чим більше в системі елементів, тим більше можливих комбінацій дефектів, які спричиняють виникнення помилок, що призводять до відмов.

У роботах розглянуто концепцію реактивних мікросервісів, які використовують асинхронну комунікацію та неблокуючий ввід/вивід для підвищення масштабованості та відмовостійкості. Хоча цей підхід

демонструє потенціал у підвищенні продуктивності та стійкості до збоїв, його впровадження вимагає ретельного проектування та використання спеціалізованих інструментів.

Згідно з роботою, методи забезпечення відмовостійкості можна розділити на реактивні, проактивні та стійкі. Реактивні методи, такі як повторні спроби та вимикачі ланцюгів, спрямовані на відновлення після виникнення збою. Проактивні методи, такі як балансування навантаження та реплікація, прагнуть запобігти збоям шляхом розподілу навантаження та створення резервних копій. Стійкі методи, що використовують машинне навчання та штучний інтелект, динамічно прогнозують та запобігають збоям у мінливих середовищах.

Робота пропонує більш детальну класифікацію механізмів відмовостійкості, виділяючи чотири ключові області: надлишковість даних та захист, стійкість системи і сервісу, моніторинг і відновлення, а також експлуатаційні та проектні практики. Ця класифікація дозволяє систематизувати різноманітні підходи та інструменти, що використовуються для забезпечення відмовостійкості мікросервісів.

У роботах розглядаються питання тестування мікросервісів. Автори пропонують фреймворк MicroHive для автоматизованого функціонального та надійнісного тестування, який дозволяє виявляти помилки та причинно-наслідкові зв'язки у ланцюгах збоїв. У статті досліджується проблема інтеграційного тестування безсерверних систем та пропонується удосконалений підхід, який дозволяє значно скоротити час тестування.

Інженерія хаосу (chaos engineering) є перспективним підходом до забезпечення відмовостійкості, який передбачає навмисне внесення помилок у систему для перевірки її здатності до відновлення. Цей підхід дозволяє виявити приховані проблеми та покращити стійкість системи до збоїв, проте вимагає ретельного планування та контролю.

У дослідженнях вивчається використання платформи Kubernetes та методу вимикачів для підвищення надійності мікросервісів. Автори вказують на необхідність розробки моделі, яка б дозволила визначати оптимальні методи забезпечення відмовостійкості для конкретних мікросервісних застосунків.

У роботі порівнюються два підходи до внесення помилок у систему: підхід посіву помилок (модель Міллса) та підхід ін'єкції помилок. Автори дослідження пропонують набір інструментів FaultSee для відтворюваного внесення помилок у розподілені системи, що дозволяє краще оцінити вплив несправностей.

У статті аналізуються переваги та недоліки використання контрольних точок в розподілених обчисленнях, а також проводиться порівняння існуючих алгоритмів контрольних точок.

Аналіз літератури показав, що проблема відмовостійкості мікросервісного програмного забезпечення є комплексною та багатогранною. [1] Незважаючи на значний прогрес у цій області, залишається багато відкритих питань, зокрема щодо розробки комплексних підходів, які враховують специфіку мікросервісної архітектури та вимоги до системи, а також щодо створення інструментів для автоматизованого виявлення та виправлення помилок.

Опис методу

Детальний опис вашого методу автоматизованого функціонального та відмовостійкого тестування.

Потреба в методі

Сучасні хмарні безсерверні мікросервісні системи характеризуються високою складністю, динамічністю та розподіленістю. Це створює значні виклики для забезпечення їх надійності та стійкості до збоїв. Традиційні методи тестування часто не здатні впоратися з цими викликами, оскільки не враховують специфіку безсерверного середовища та розподіленої архітектури.

Основні проблеми:

- **Складність інтеграції:** Безсерверні функції часто виконуються в ізольованих контейнерах, що ускладнює тестування їх взаємодії.
- **Непередбачувані збої:** Динамічне масштабування та ефемерність безсерверних функцій можуть призводити до непередбачуваних збоїв.

- **Обмежені можливості відлагодження:** Короткочасність безсерверних функцій ускладнює процес відлагодження та виявлення причин збоїв.

Задачі методу:

Для вирішення цих проблем потрібен новий підхід до тестування, який би забезпечував:

- **Комплексну перевірку:** Охоплення всіх рівнів тестування, від модульного до системного, з урахуванням специфіки безсерверних систем.
- **Раннє виявлення проблем:** Виявлення потенційних проблем на ранніх стадіях розробки, що спрощує їх виправлення та знижує витрати.
- **Підвищення стійкості:** Перевірка стійкості системи до збоїв та непередбачуваних ситуацій.
- **Оптимізацію використання ресурсів:** Виявлення та усунення проблем, які можуть призвести до непередбачуваних витрат на хмарні ресурси.

Опис методу з удосконаленнями

Для вирішення поставлених задач пропонується метод "Пентагон тестування", який є комплексним підходом до тестування безсерверних мікросервісних систем. Метод включає наступні етапи:

1. **Модульне тестування:** Перевірка окремих безсерверних функцій на коректність виконання їх основної логіки.
2. **Локальне інтеграційне тестування:** Перевірка взаємодії між функціями в межах одного сервісу з використанням моків та стабів для імітації залежностей.
3. **Глобальне інтеграційне тестування:** Розгортання системи в середовищі, наближеному до продуктивного, та перевірка взаємодії між сервісами з використанням реальних даних та навантажень.
4. **Тестування відмовостійкості:** Симуляція різних видів збоїв (відмова функцій, мережеві затримки, втрата даних) та перевірка здатності системи до відновлення.

5. **Хаос-тестування:** Внесення непередбачуваних змін в роботу системи (наприклад, випадкове вимкнення функцій, зміна конфігурації) для перевірки її стійкості та здатності до адаптації.

Удосконалення методу:

- **Автоматизація:** Автоматизація всіх етапів тестування для підвищення ефективності та зниження витрат.
- **Інтеграція з CI/CD:** Вбудування "Пентагону тестування" в процес CI/CD для забезпечення безперервного контролю якості.
- **Моніторинг:** Використання систем моніторингу для збору даних про роботу системи в реальному часі та виявлення потенційних проблем.
- **Хаос-інженерія:** Застосування принципів хаос-інженерії для симуляції різних непередбачуваних ситуацій та перевірки стійкості системи.

Найкращий аналог та порівняння

Найкращим аналогом для порівняння з "Пентагоном тестування" є підхід, запропонований в роботі **"An extensible fault tolerance testing framework for microservice-based cloud applications"**. Обидва підходи передбачають використання фреймворків для автоматизованого тестування відмовостійкості, але "Пентагон тестування" має ряд переваг:

- **Більш широке охоплення:** Включає хаос-тестування та тестування відмовостійкості, що дозволяє перевірити більш широкий спектр потенційних проблем.
- **Інтеграція з CI/CD:** Забезпечує тіснішу інтеграцію з процесом CI/CD, що дозволяє отримувати зворотний зв'язок про якість коду та стійкість системи на кожному етапі розробки.
- **Адаптивність:** Може бути налаштований під специфіку конкретного проекту, включаючи вибір інструментів та типів тестів.

Метрики для порівняння ефективності

Для порівняння ефективності "Пентагону тестування" з аналогом будуть вимірюватися наступні метрики:

- **MTBF (Mean Time Between Failures):** середній час між відмовами.

- **Failure Rate:** частота відмов.
- **Test Coverage:** покриття коду тестами.
- **System Availability:** доступність системи.
- **Failure Avoidance:** кількість запобігнутих відмов.
- **Mean Fault Notification Time:** середній час сповіщення про відмову.
- **Fault Correction:** час, необхідний для виправлення відмови.
- **Час відповіді:** час, необхідний для обробки запиту безсерверною функцією.
- **Використання ресурсів:** обсяг споживаних ресурсів (пам'ять, процесорний час) безсерверними функціями.
- **Масштабованість:** здатність системи масштабуватися для обробки зростаючого навантаження.

Вимірювання цих метрик дозволить оцінити ефективність "Пентагону тестування" в порівнянні з аналогом та зробити висновки про його переваги та недоліки.

Доведення ефективності підходу "Пентагон тестування"

Для доведення ефективності підходу "Пентагон тестування" буде проведено експериментальне дослідження, яке включатиме наступні етапи:

1. **Розробка тестового середовища:**
    - Створення репрезентативної безсерверної мікросервісної системи, що буде використовуватися для проведення експериментів. Система повинна мати достатню складність та включати різні типи функцій та взаємодій.
    - Розгортання системи в хмарному середовищі
    - Впровадження інструментів моніторингу та збору метрик для відстеження роботи системи.
2. **Впровадження підходів до тестування:**
    - **Базова система:** Впровадження тестування за стратегією "соти тестування" з використанням фреймворку з роботи "An extensible fault tolerance testing framework for microservice-based cloud applications".

- **Експериментальна система:** Впровадження "Пентагону тестування" з урахуванням всіх удосконалень (автоматизація, інтеграція з CI/CD, моніторинг, хаос-інженерія).

3. **Проведення стрес-тестування:**

    - Створення реалістичних сценаріїв навантаження, що імітують роботу системи в реальних умовах.
    - Проведення стрес-тестування обох систем (базової та експериментальної) з однаковим навантаженням.
    - **Тестування з ін'єкцією помилок:**
        i. Визначення типових помилок, які можуть виникати в безсерверних системах (наприклад, помилки мережі, помилки доступу до баз даних, помилки виконання функцій).
        ii. Використання інструментів для ін'єкції помилок (наприклад, Chaos Monkey, Gremlin, AWS Fault Injection Simulator) для симуляції цих помилок під час роботи системи.
        iii. Збір даних про поведінку системи в умовах ін'єкції помилок (час відновлення, кількість успішно оброблених запитів, кількість помилок).

Аналіз результатів:

- Порівняння значень метрик для базової та експериментальної систем, отриманих під час стрес-тестування та тестування з ін'єкцією помилок.
- Статистичний аналіз даних для визначення достовірності отриманих результатів.
- Висновки про ефективність підходу "Пентагон тестування" в порівнянні з базовою системою, з урахуванням результатів обох видів тестування.

Архітектура системи тестування.

Система адміністрування студентів буде побудована на безсерверній архітектурі з використанням хмарних сервісів AWS. Вона складатиметься з трьох основних мікросервісів:

- Сервіс студентів (Student Service)
- Сервіс груп (Group Service)
- Сервіс звітів (Report Service)

Кожен сервіс буде реалізований за допомогою AWS Lambda функцій та взаємодіятиме з власною базою даних DynamoDB. API Gateway буде використовуватися для обробки зовнішніх запитів та маршрутизації їх до відповідних сервісів. Додатково будуть використовуватися SQS для асинхронної обробки запитів та X-Ray для трасування запитів та аналізу продуктивності.

Детальний опис компонентів

1. Сервіс студентів (Student Service)

- **Функціонал:**
  - Створення, читання, оновлення та видалення даних студентів (CRUD).
  - Пошук студентів за різними критеріями (прізвище, ім'я, група, рік навчання).
- **Технології:**
  - AWS Lambda: для реалізації функцій, що обробляють запити.
  - DynamoDB: для зберігання даних студентів (окрема таблиця).
  - API Gateway: для обробки зовнішніх запитів та маршрутизації їх до Lambda функцій.

2. Сервіс груп (Group Service)

- **Функціонал:**
  - Створення, читання, оновлення та видалення груп.
  - Додавання та видалення студентів з груп.
  - Отримання списку студентів в групі.
- **Технології:**
  - AWS Lambda: для реалізації функцій, що обробляють запити.
  - DynamoDB: для зберігання даних груп та зв'язків студентів з групами (окрема таблиця).
  - API Gateway: для обробки зовнішніх запитів та маршрутизації їх до Lambda функцій.

3. Сервіс звітів (Report Service)

- **Функціонал:**
  - Генерація звітів про студентів та групи (наприклад, список студентів за групами, статистика успішності).
- **Технології:**

    - AWS Lambda: для реалізації функцій, що генерують звіти.
    - DynamoDB: для отримання даних для звітів.
    - S3: для зберігання згенерованих звітів.
    - API Gateway: для обробки зовнішніх запитів та маршрутизації їх до Lambda функцій.
    - SQS: для отримання повідомлень про запити на генерацію звітів (асинхронна обробка).

4. API Gateway

- **Функціонал:**
    - Обробка вхідних запитів від клієнтів.
    - Аутентифікація та авторизація запитів (опціонально).
    - Маршрутизація запитів до відповідних Lambda функцій в залежності від типу запиту та URL.
    - Трансформація запитів та відповідей (опціонально).

5. DynamoDB

- **Функціонал:**
    - Зберігання даних студентів, груп та зв'язків між ними.
    - Забезпечення високої доступності та масштабованості.

6. CloudWatch

- **Функціонал:**
    - Моніторинг роботи Lambda функцій та API Gateway.
    - Збір логів та метрик.
    - Налаштування сповіщень про помилки та критичні події.

7. AWS SAM (Serverless Application Model)

- **Функціонал:**
    - Визначення інфраструктури системи (Lambda функції, API Gateway, DynamoDB, SQS) в коді.
    - Спрощення розгортання та керування системою.

8. GitLab CI/CD

- **Функціонал:**
    - Автоматизація процесу розробки та розгортання.
    - Інтеграція з AWS SAM для автоматичного розгортання системи в хмарному середовищі.

9. AWS SQS (Simple Queue Service)

- **Функціонал:**
    - Асинхронна обробка запитів на генерацію звітів.
    - Сервіс звітів отримує повідомлення про запити з черги SQS та генерує звіти асинхронно.

- Це дозволяє зменшити час відгуку API та підвищити стійкість системи до збоїв.

10. AWS X-Ray

- **Функціонал:**
  - Трасування запитів, що проходять через систему.
  - Аналіз продуктивності кожного компонента системи.
  - Виявлення вузьких місць та проблем з продуктивністю.

Взаємодія компонентів

1. Клієнт надсилає запит до API Gateway.
2. API Gateway аутентифікує та авторизує запит (опціонально).
3. API Gateway маршрутизує запит до відповідної Lambda функції в залежності від типу запиту та URL.
4. Lambda функція обробляє запит, взаємодіючи з DynamoDB для отримання або зміни даних.
5. При необхідності Lambda функція може відправляти запити до інших Lambda функцій (наприклад, сервіс груп може відправляти запити до сервісу студентів).
6. Для асинхронних операцій (наприклад, генерація звітів) Lambda функція відправляє повідомлення в чергу SQS.
7. Сервіс звітів отримує повідомлення з черги SQS та генерує звіт асинхронно.
8. Результат виконання Lambda функції повертається клієнту через API Gateway.
9. X-Ray трасує запити та збирає дані про продуктивність кожного компонента.
10. CloudWatch збирає логи та метрики про роботу системи.

Алгоритми та інструменти.

Експериментальне дослідження

Опис тестового середовища та процедури тестування.

Чудово! Давайте детально розробимо стратегію реалізації та дослідження, включаючи створення базової версії системи з використанням "соти тестування", а потім удосконалення її з використанням підходу "Пентагон тестування".

Етап 1: Створення базової версії системи

1. **Проектування та розробка:**
   - Детальне проектування архітектури системи з урахуванням трьох мікросервісів (Student Service, Group Service, Report Service), API Gateway, DynamoDB, SQS, X-Ray та SNS.
   - Розробка Lambda функцій для кожного сервісу з базовим функціоналом CRUD, пошуку, додавання/видалення з груп та генерації звітів.

- Визначення схеми даних в DynamoDB для кожного сервісу.
- Налаштування API Gateway для маршрутизації запитів до відповідних Lambda функцій.
- Створення шаблону AWS SAM для визначення інфраструктури системи.

2. **Впровадження "соти тестування":**

   - Написання юніт-тестів для кожної Lambda функції, що перевіряють коректність виконання бізнес-логіки.
   - Написання інтеграційних тестів для перевірки взаємодії між Lambda функціями в межах одного сервісу.
   - Написання компонентних тестів для перевірки взаємодії між сервісами.
   - Використання моків та стабів для імітації залежностей під час тестування.

3. **Автоматизація з GitLab CI/CD:**

   - Налаштування CI/CD pipeline в GitLab для автоматизації процесу збірки, тестування та розгортання системи.
   - Інтеграція з AWS SAM для автоматичного розгортання системи в хмарному середовищі AWS.

4. **Розгортання та налаштування моніторингу:**

   - Розгортання системи в AWS.
   - Налаштування CloudWatch для моніторингу роботи Lambda функцій, API Gateway та інших компонентів.
   - Налаштування X-Ray для трасування запитів та аналізу продуктивності.
   - Налаштування SNS для отримання сповіщень про критичні помилки.

5. **Впровадження багів:**

   - Впровадження двох багів даних в систему для перевірки ефективності підходів до тестування.

Етап 2: Тестування базової версії

1. **Проведення 72-годинного тестування:**
   - Запуск системи та генерація навантаження, що імітує реальні умови використання.
   - Збір даних про роботу системи за допомогою CloudWatch та X-Ray.
   - Вимірювання метрик надійності (MTBF, Failure Rate, System Availability, Failure Avoidance, Mean Fault Notification Time, Fault Correction, час відповіді, використання ресурсів, масштабованість).

Етап 3: Удосконалення системи з "Пентагоном тестування"

1. **Впровадження "Пентагону тестування":**
   - Додавання етапів тестування відмовостійкості та хаос-тестування.
   - Автоматизація всіх етапів тестування.
   - Інтеграція з CI/CD pipeline.

- Використання принципів хаос-інженерії для симуляції непередбачуваних ситуацій.

Етап 4: Тестування удосконаленої версії

1. **Проведення 72-годинного тестування:**
   - Запуск удосконаленої системи та генерація такого ж навантаження, як і для базової версії.
   - Збір даних про роботу системи за допомогою CloudWatch та X-Ray.
   - Вимірювання тих самих метрик надійності, що і для базової версії.

Етап 5: Аналіз результатів

1. **Порівняння метрик:**
   - Порівняння значень метрик, отриманих для базової та удосконаленої версій системи.
   - Статистичний аналіз даних для визначення достовірності результатів.
2. **Висновки:**
   - Формулювання висновків про ефективність підходу "Пентагон тестування" в порівнянні з "сотою тестування".
   - Оцінка переваг та недоліків кожного підходу.

Додаткові аспекти:

- **Документування:** Ретельне документування всіх етапів дослідження, включаючи проектування, розробку, тестування та аналіз результатів.
- **Версійний контроль:** Використання Git для відстеження змін в коді та конфігурації системи.
- **Відкритий код:** Розміщення коду системи та інфраструктури у відкритому доступі для забезпечення прозорості та можливості відтворення результатів дослідження.

Висновки:

Запропонована стратегія дозволить провести комплексне дослідження ефективності підходу "Пентагон тестування" для забезпечення надійності безсерверних мікросервісних систем. Отримані результати допоможуть оцінити переваги та недоліки підходу та зробити висновки про його застосовність в реальних проектах.

Аналіз результатів експериментів.

Порівняння з іншими методами.

Обговорення результатів

Інтерпретація отриманих результатів.

Обмеження методу та напрямки подальших досліджень.

Висновки

Основні результати дослідження.

Внесок у вирішення проблеми відмовостійкості мікросервісного ПЗ.

Список літератури (оформлений згідно ДСТУ 8302:2015)