

# ЗАБЕЗПЕЧЕННЯ ВІДМОВСТІЙКОСТІ ХМАРНИХ СЕРЕДОВИЩ МЕТОДАМИ GITOPS

Ільїнов Кирило Миколайович

*студент групи МПМ-1-25-1.4д,*

*Київського столичного університету імені Бориса Грінченка,*

*науковий керівник – д.т.н., проф. Бушма О.В.*

## ВСТУП

**Актуальність і постановка проблеми.** Хмарні середовища складаються з великої кількості взаємопов'язаних компонентів: серверів, контейнерів, мережових правил і файлів налаштувань. Збій будь-якого з них може зупинити роботу сервісу. Причинами збоїв бувають програмні помилки, відмови обладнання та помилкові ручні зміни, які вносять адміністратори [1; 2].

Окремою загрозою є дрейф конфігурації (від англ. configuration drift). Так називають поступове розходження між задокументованим і фактичним станом системи. Дрейф виникає, коли зміни вносять вручну безпосередньо на сервері, оминаючи процедуру перевірки. З часом такий сервер стає унікальним, і відновити його після аварії важко [2].

Традиційне забезпечення відмовостійкості спирається на моніторинг подій та ручні інструкції для інженерів підтримки [3]. Інженер має помітити сповіщення, знайти причину збою і виконати відновлення. Увесь цей час сервіс може бути недоступним. Досвід великих компаній показує, що ручна повторювана робота з експлуатації погано масштабується і є джерелом помилок [3].

Отже, проблема полягає в тому, що час відновлення хмарного середовища після збою залежить від швидкості реакції людини, а фактичний стан системи може непомітно відходити від еталона.

**Аналіз останніх досліджень і публікацій.** Резервне копіювання дає змогу відновити дані після аварії. Проте воно не відстежує поточну конфігурацію системи і не повертає її автоматично до правильного стану [2]. Для цього потрібен постійний еталон, з яким можна порівнювати фактичний стан.

Методологія GitOps (від англ. Git та Operations – експлуатація) пропонує використовувати таким еталоном репозиторій системи контролю версій Git [4]. Принципи OpenGitOps вимагають, щоб програмні агенти безперервно спостерігали за фактичним станом системи і намагалися привести його до бажаного [5]. Ця ідея походить від архітектури системи Kubernetes. У ній окремі контролери циклічно порівнюють фактичний стан ресурсів із заданим і усувають різницю [1].

Kubernetes уже має вбудовані засоби відновлення. Контролер розгортання (Deployment) підтримує задану кількість копій застосунку [6]. Перевірки працездатності (від англ. liveness probe) дають змогу автоматично перезапустити контейнер, який перестав відповідати [7]. Інструмент Argo CD додає режим самовідновлення selfHeal, який повертає змінені вручну ресурси до стану з репозиторію [8].

Таким чином, у розглянутих джерелах [1; 5–8] ці механізми описано окремо, кожен для свого класу збоїв. Питання їх поєднання в єдину модель відмовостійкості, де кожен тип збою має свій автоматичний механізм відновлення, висвітлено недостатньо.

Актуальність дослідження зумовлена суперечністю між зростаючою складністю хмарних систем і повільною реакцією ручного адміністрування на аварії. Відсутність цілісної моделі, яка поєднує вбудовані механізми Kubernetes і безперервне узгодження з репозиторієм, зумовила вибір теми роботи.

**Мета дослідження** – розробити архітектуру дворівневого автоматичного відновлення хмарного середовища на основі методології GitOps, яка повертає середовище до еталонного стану після програмних збоїв і помилкових ручних змін без участі людини.

**Наукова новизна.** Набув подальшого розвитку підхід до забезпечення відмовостійкості хмарних середовищ. Запропоновано дворівневу модель відновлення. На першому рівні вбудовані контролери Kubernetes усувають збої виконання, тобто падіння процесів і контейнерів. На другому рівні агент GitOps усуває відхилення конфігурації від версіонованого репозиторію. На відміну від резервного копіювання, така модель відновлює саме конфігурацію системи і робить це безперервно, а не за розкладом.

**Об’єкт дослідження** – процеси забезпечення відмовостійкості та самовідновлення хмарної IT-інфраструктури.

**Предмет дослідження** – методи та засоби автоматичного відновлення хмарних середовищ на основі методології GitOps.

Для досягнення мети поставлено такі **завдання**:

- 1) сформувати дворівневу модель відновлення, у якій кожному класу збоїв відповідає свій автоматичний механізм;
- 2) розробити декларативні конфігурації: маніфест застосунку з перевірками працездатності та політику автоматичної синхронізації з режимом самовідновлення;
- 3) отримати експериментальні оцінки поведінки системи у двох сценаріях збоїв.

## МЕТОДИ ТА ЗАСОБИ ДОСЛІДЖЕННЯ

Для досягнення мети використано метод класифікації (для розподілу збоїв за типами), метод архітектурного моделювання та експеримент з імітацією збоїв на тестовому стенді.

Уся конфігурація робочого середовища зберігається в репозиторії Git у вигляді декларативних маніфестів. Маніфест описує бажаний результат, а не послідовність команд [5]. Тому репозиторій є єдиним джерелом достовірної інформації про те, яким має бути середовище [4].

Запропоновану дворівневу модель показано на рис. 1. Обидва рівні працюють за однаковим принципом замкненого циклу: спостерегти фактичний стан, порівняти з бажаним, усунути різницю [1]. Різниця між рівнями полягає в тому, які збої вони виявляють.

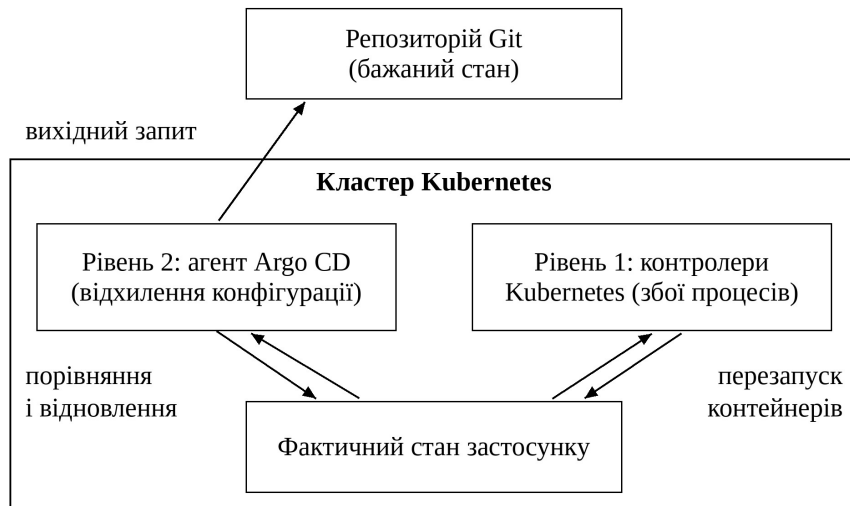


Рис. 1. Дворівнева модель автоматичного відновлення хмарного середовища

Перший рівень утворюють вбудовані контролери Kubernetes. Вони реагують на збої виконання. Якщо процес усередині контейнера аварійно завершився або перестав відповідати, контролер перезапускає контейнер. Якщо зник один з екземплярів застосунку, контролер створює новий [6; 7].

Другий рівень утворює агент Argo CD, який працює всередині кластера. Він реагує на відхилення конфігурації. Агент періодично зчитує бажаний стан із репозиторію та порівнює його з фактичним станом ресурсів у кластері. Якщо хтось вручну змінив або видалив ресурс, агент повертає його до стану, описаного в Git [8].

Поведінку першого рівня задає маніфест застосунку, наведений на рис. 2. Цей маніфест також зберігається в репозиторії, тому другий рівень захищає від випадкових змін і сам опис першого рівня.

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: main-web-service
  namespace: production
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
        - name: web
          image: registry.example.org/web:1.0
          livenessProbe:
            httpGet:
              path: /healthz
              port: 8080
              periodSeconds: 5

```

Рис. 2. Маніфест застосунку з параметрами відмовостійкості

Параметр `replicas` задає три копії застосунку. Якщо одна копія виходить з ладу, дві інші продовжують обслуговувати користувачів, поки контролер створює заміну [6]. Блок `livenessProbe` описує перевірку працездатності. Кожні 5 секунд Kubernetes надсилає запит на службову адресу `/healthz`. Якщо контейнер не відповідає, його перезапускають [7].

Поведінку другого рівня задає політика синхронізації в описі застосунку Argo CD. У ній увімкнено автоматичну синхронізацію з параметрами `selfHeal` і `prune` [8]. Параметр `selfHeal` (самовідновлення) повертає змінені вручну ресурси до еталона. Параметр `prune` (видалення зайвого) прибирає з кластера ресурси, яких уже немає в репозиторії [8].

Будь-яке виправлення конфігурації після збою виконується тільки через запит на злиття змін у репозиторії. Це забезпечує перевірку кожної зміни іншими інженерами та повну історію дій персоналу [4].

Стійкість архітектури перевірено на тестовому кластері Kubernetes під керуванням операційної системи Linux. Експеримент складався з двох сценаріїв. У першому сценарії примусово завершували роботу критичних процесів застосунку. У другому сценарії моделювали людську помилку: вручну видаляли ресурси конфігурації безпосередньо в кластері.

## РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ

В обох сценаріях система повернулася до робочого стану без втручання людини. У першому сценарії відновлення виконали контролери Kubernetes: аварійно завершені контейнери було перезапущено автоматично.

У другому сценарії спрацював другий рівень моделі. Агент Argo CD виявив розбіжність між кластером і репозиторієм за 3 секунди. Повне повернення до еталонного стану тривало 14 секунд. Розподіл механізмів відновлення за типами збоїв наведено в табл. 1.

Таблиця 1

Розподіл механізмів відновлення за типами збоїв

| Тип збою                          | Механізм відновлення                             | Без автоматизації                           |
|-----------------------------------|--------------------------------------------------|---------------------------------------------|
| Аварійне завершення процесу       | Перезапуск контейнера контролером Kubernetes     | Ручний перезапуск після сповіщення          |
| Зависання застосунку              | Перевірка працездатності і перезапуск контейнера | Пошук причини в журналах, ручний перезапуск |
| Втрата одного екземпляра          | Створення нового екземпляра до заданої кількості | Ручне розгортання нового екземпляра         |
| Ручна зміна або видалення ресурсу | Режим <code>selfHeal</code> агента Argo CD       | Зміна залишається непоміченою               |
| Ресурс видалено з репозиторію     | Режим <code>prune</code> агента Argo CD          | Застарілий ресурс залишається в кластері    |

Дані таблиці показують, що кожен клас збоїв, розглянутий у роботі, має власний автоматичний механізм відновлення. Перший рівень захищає від збоїв

виконання, а другий – від дрейфу конфігурації. Разом вони замінюють ручні інструкції для інженерів підтримки.

Запропонована модель має обмеження. Вона не захищає від помилки, яку внесли безпосередньо в репозиторій: агент відтворить помилкову конфігурацію так само точно, як правильну. Тому якість відновлення залежить від перевірки змін перед злиттям. Також модель не замінює резервного копіювання даних, оскільки відновлює конфігурацію, а не вміст баз даних.

## ВИСНОВКИ

У роботі показано, що ручне реагування на аварії робить час відновлення залежним від людини, а ручні зміни спричиняють дрейф конфігурації. Для розв'язання цієї проблеми розроблено архітектуру дворівневого автоматичного відновлення на основі методології GitOps.

При виконанні дослідження, по-перше, сформовано дворівневу модель відновлення: контролери Kubernetes усувають збої виконання, а агент Argo CD – відхилення конфігурації. По-друге, розроблено декларативні конфігурації: маніфест застосунку з трьома копіями та перевіркою працездатності і політику синхронізації з режимами selfHeal і prune. По-третє, отримано експериментальні оцінки: в обох сценаріях система відновилася без участі людини, а після ручного видалення ресурсів повне відновлення тривало 14 секунд.

Таким чином, запропоноване архітектурне рішення дворівневого автоматичного відновлення хмарного середовища на основі методології GitOps дозволило успішно реалізувати автоматичне повернення хмарного середовища до еталонного стану після програмних збоїв і помилкових ручних змін.

## ДЖЕРЕЛА

1. Burns B., Grant B., Oppenheimer D., Brewer E., Wilkes J. Borg, Omega, and Kubernetes. Communications of the ACM. 2016. Vol. 59, No. 5. P. 50–57. DOI: <https://doi.org/10.1145/2890784>.
2. Morris K. Infrastructure as Code: Dynamic Systems for the Cloud Age. 2nd ed. Sebastopol : O'Reilly Media, 2020.
3. Beyer B., Jones C., Petoff J., Murphy N. R. Site Reliability Engineering: How Google Runs Production Systems. Sebastopol : O'Reilly Media, 2016.
4. Beetz F., Harrer S. GitOps: The Evolution of DevOps? IEEE Software. 2022. Vol. 39, No. 4. P. 70–75. DOI: <https://doi.org/10.1109/MS.2021.3119106>.
5. GitOps Principles v1.0.0 / OpenGitOps. 2021. URL: <https://opengitops.dev/> (дата звернення: 08.10.2026).
6. Deployments. Kubernetes Documentation. URL: <https://kubernetes.io/docs/concepts/workloads/controllers/deployment/> (дата звернення: 08.10.2026).
7. Configure Liveness, Readiness and Startup Probes. Kubernetes Documentation. URL: <https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/> (дата звернення: 08.10.2026).
8. Automated Sync Policy. Argo CD Documentation. URL: [https://argo-cd.readthedocs.io/en/stable/user-guide/auto\\_sync/](https://argo-cd.readthedocs.io/en/stable/user-guide/auto_sync/) (дата звернення: 08.10.2026).