УДК 004.4'2 + 004.75

В. В. Кінчур, І. В. Стеценко

АРХІТЕКТУРА СЕРВЕРА АВТОМАТИЗАЦІЇ ДЛЯ ОРКЕСТРАЦІЇ ЗАДАЧ У ХМАРНОМУ СЕРЕДОВИЩІ

Анотація: Сьогодні більшість інструментів автоматизації реалізує виконання процесів автоматизації в хмарі. Однак використовують агентне ПЗ, що ускладнює первинне налаштування і подальший супровід інструментів. У даному науковому дослідженні запропоновано архітектуру сервера автоматизації, що реалізує односторонню взаємодію з хмарою за допомогою нативних засобів оркестраторів контейнерів. Таке рішення не вимагає внесення змін до хмарної інфраструктури для її використання сервером автоматизації, тому спрощує налаштування сервера і зменшує кількість використовуваного обчислювального ресурсу. Архітектура є модульною, де кожна функціональна задача реалізована окремим плагіном.

Ключові слова: сервер автоматизації, хмарні обчислення, процеси автоматизації розробки і розгортання програмного забезпечення, оркестрація задач

Опис проблеми

Хмарні рішення стали невід’ємною частиною циклу розробки програмного забезпечення (ПЗ) для багатьох ІТ проєктів. Хмара надає гнучкість і зручність використання, які складно досягти у межах власної інфраструктури, що дозволяє пришвидшити розробку і масштабування ПЗ. Однак разом з тим призводить до ускладнення процесів розробки і розгортання ПЗ, що при цьому виникають. Таке ускладнення стало каталізатором створення рішень, що дозволяють інтегрувати хмарні засоби з інструментами автоматизації. Як результат, більшість сучасних інструментів автоматизації реалізують інтеграцію з хмарою. Однак кожен з них вимагає внесення постійних чи тимчасових змін до хмарної інфраструктури, що полягає у встановленні агентного (agent/runner) ПЗ в її межах [1, 2, 3, 4]. Це ускладнює первинне налаштування і подальший супровід таких інструментів. Власне тому актуальним є завдання створення сервера автоматизації, що дозволить виконання процесів розробки та розгортання ПЗ в межах хмарних середовищ без необхідності внесення змін до інфраструктури, надаючи при цьому засоби для адаптації сервера до вимог користувача.

Сервер автоматизації

Метою інструментів автоматизації є уникнення повторного виконання рутинних кроків, що виникають у ході розробки чи розгортання ПЗ. Процеси автоматизації застосовують декларативний опис цих кроків та забезпечують їх

© В. В. Кінчур, І. В. Стеценко

виконання без необхідності повторення [5]. Кількість і зміст таких процесів напряму залежить від ПЗ, в межах якого вони виникають.

Поширеним типом інструментів автоматизації є сервери автоматизації. Сервер автоматизації – це інструмент або платформа, призначена для автоматизації повторюваних процесів, що виникають в розробці програмного забезпечення, IT-операціях та інших робочих процесах. Сервер автоматизації планує, виконує й керує відповідними процесами [1].

Спільною рисою більшості серверів автоматизації є надання абстрактного засобу для опису процесів автоматизації, не вдаючись при цьому до класифікації таких процесів [1, 2, 3, 4]. Це пояснюється передусім тим, що складові процесу залежать від ПЗ, в межах якого вони виникають, а тому доцільніше покласти задачу опису процесів на користувача, якому краще відома специфіка цільового ПЗ. Такий опис досягається за рахунок представлення програмних задач (job/task) [1, 2, 3, 4]. Опис задачі включає три основних елементи:

− тригер – розклад, подія, або мануальний запуск, що призводять до виконання задачі;

− середовище – операційна система, контейнер, віртуальна машина тощо, в межах якої виконується задача;

− кроки – послідовність дій, що потрібно виконати в межах задачі.

У той час як засоби для опису тригеру й кроків виконання є схожими для різних інструментів автоматизації, опції для середовища виконання часто відрізняються. Виділяють щонайменше виконання задач у межах локального, хмарного чи гібридного середовища.

Процеси автоматизації в хмарі

Незалежно від мови опису чи складності кроків, довільний процес автоматизації зводиться до виконання послідовності дій, що є зрозумілими для середовища, в межах якого цей процес розгортається. Тобто, інтеграція з хмарою полягає у віддаленому виконанні коду у відповідності до визначених в описі кроків процесу автоматизації за допомогою хмарного обчислювального ресурсу.

Залежно від способу взаємодії між хмарою і сервером автоматизації, можна виділити три основних підходи:

− перманентні вузли [6];

− хмарні функції [7, 8];

− ефемерні контейнери [9, 10].

Підхід з перманентними вузлами передбачає попереднє розгортання в хмарі вузла, що надає серверу автоматизації програмний інтерфейс для передачі кроків

 ISSN 1560-8956

процесу автоматизації, які необхідно виконати [6]. Взаємодія між сервером і вузлом зазвичай забезпечується за допомогою агентного ПЗ, що повинно бути встановлено в межах вузла. Такий підхід є гнучким, однак вимагає дій з боку користувача, що полягають у налаштуванні чи подальшому супроводі вузла та агентого ПЗ. При цьому вузол є активним постійно, що призводить до простоювання обладнання.

У випадку з хмарними функціями сервер автоматизації попередньо створює хмарну функцію, що містить кроки процесу автоматизації. При настанні умови виконання задачі, сервер автоматизації викликає відповідну функцію [7, 8]. Цей підхід є найменш ресурсовитратним і не вимагає жодних дій з боку користувача. Однак хмарні функції надають обмежені засоби для налаштування середовища виконання, що часто є недостатньо гнучким для реалізації процесів автоматизації.

У межах підходу з ефемерними контейнерами сервер автоматизації динамічно створює в хмарі контейнер, що містить кроки процесу автоматизації й знищується після завершення виконання задачі [9, 10]. Цей підхід є де-факто стандартним для виконання коду у хмарі, адже надає достатню гнучкість налаштування середовища і є простішим для налаштування, ніж підхід з вузлами. Власне тому ефемерні контейнери розглядаються у даному дослідженні.

Іншою проблемою, що виникає при інтеграції, є широкий контекст поняття "хмари". Можна виділити класифікацію хмари за типом (приватна, публічна, гібридна), моделлю виконання (IAAS, PAAS, SAAS), вендором (AWS, GCP, Azure) тощо. Інтеграція у контексті кожного з класів вимагає специфічних кроків, що ускладнює імплементацію універсального рішення. Зважаючи не це, доречно скористатися іншою перевагою контейнерів, а саме наявністю оркестраторів контейнерів – інструментів, що дозволяють узагальнити взаємодію з хмарою. Прикладами таких інструментів є Kubernetes, Docker Swarm, AWS ECS, Apache Mesos, HashiCorp Nomad. Основною перевагою оркестраторів контейнерів є можливість абстрагуватися від внутрішнього представлення хмари і взаємодіяти з нею як з єдиною обчислювальною одиницею – кластером [11].

Аналіз існуючих рішень

До аналогів можна віднести такі інструменти, як Jenkins, GitHub Actions, GitLab CI/CD, Bitbucket Pipelines тощо. Кожен з них реалізує інтеграцію з хмарними середовищами. Наприклад, Jenkins досягає цього за допомогою використання agents, що розгортаються в довільному хмарному чи локальному середовищі [1], а в платформах автоматизації виконання задач відбувається у межах приватної хмарної інфраструктури за замовчуванням, з наданням при цьому можливості розгортання задач у межах зовнішньої інфраструктури користувача за допомогою self-hosted runners [2, 3, 4].

Зазначимо, що кожен з аналогів вимагає внесення змін до хмарної інфраструктури [1, 2, 3, 4]. GitHub Actions вимагає попереднього встановлення Actions Runner в межах кожної хмарної одиниці [2], Jenkins – Jenkins Agent [1], GitLab CI/CD – GitLab Runner [3], а BitBucket Pipelines – Runner [4]. Якщо у випадку контейнерів встановлення може бути виконано безпосередньо у момент запуску задачі, то вузли потребують його попереднього встановлення. Інакше кажучи, зміни до інфраструктури вносяться попередньо на тривалий період або динамічно на період виконання задачі.

Разом з тим кожен з аналогів реалізує інтеграцію з оркестраторами контейнерів, зокрема Kubernetes [1, 2, 3, 4]. Однак, як і у випадку з агентним ПЗ, вимагають внесення змін до кластеру. GitHub Actions вимагає встановлення Custom Resource Definition (CRD) Actions Runner Controller (ARC) [2], що керує робочими вузлами в межах кластера. Натомість GitLab CI/CD й BitBucket Pipelines вимагають розгортання агентного ПЗ у вигляді вузла в кластері [3, 4]. Тобто, всі платформи автоматизації потребують попереднього налаштування кластеру, що користувач повинен зробити самостійно. У той же час Jenkins використовує інший підхід, а саме – встановлює Jenkins Agent самостійно при кожному запуску контейнера задачі [1]. Тобто, у такому випадку користувач не повинен вносити довготривалі зміни до кластеру.

Основним недоліком такого підходу є те, що він ускладнює міграцію між інструментами автоматизації, адже кожен з них вимагає внесення інших змін до хмарної інфраструктури. Крім цього, агентне ПЗ використовує додаткові обчислювальні ресурси хмари.

Модель взаємодії з хмарою

У даному дослідженні розглядається рішення, що покликано спростити розгортання процесів автоматизації в хмарі шляхом уникнення використання агентного ПЗ в межах хмарної інфраструктури. Розроблений сервер автоматизації реалізує інтеграцію з хмарою за допомогою ефемерних контейнерів, що розгортаються засобами оркестраторів контейнерів. Пряма комунікація з окремими вузлами у хмарі не відбувається. За таких умов задача управління хмарними ресурсами покладається на оркестратор, який виступає у ролі проміжного шару між сервером автоматизації та хмарною інфраструктурою. Тобто сервер автоматизації виконує запити для отримання обчислювальних ресурсів у межах хмари, які надалі будуть використані для розгортання програмної задачі, а оркестратор ці ресурси виділяє. Для того, щоб уникнути внесення змін до інфраструктури, взаємодія з хмарою забезпечується лише за рахунок нативних засобів оркестраторів контейнерів.

Незважаючи на те, що різні оркестратори контейнерів мають різний API, запропонована модель може бути використаною для більшості існуючих рішень. Це

 ISSN 1560-8956

пояснюється передусім тим, що сервер автоматизації ставить лише базові вимоги до оркестраторів, які більшість існуючих інструментів реалізують. Вимоги є наступними:

− створення контейнера, у межах якого будуть виконані кроки процесу автоматизації;

− моніторинг стану контейнера (ініціалізується, виконується, чи виконання завершено);

− отримання логів і, якщо можливо, метрик.

У межах запропонованої моделі користувач не взаємодіє з хмарою напряму. Натомість запити надходять лише до сервера автоматизації, який надалі забезпечує комунікацію з середовищем, що використовується для виконання задач. Середовище при цьому може бути представленим як у вигляді хмари, так і вузла, на якому розгорнуто сервер автоматизації. Інакше кажучи, розроблене рішення може використовувати як локальні, так і зовнішні обчислювальні ресурси для виконання задач. При цьому може взаємодіяти з кількома такими середовищами одночасно, надаючи користувачу можливість обрати цільове середовище на рівні кожної задачі.

Таким чином, між користувачем і хмарною інфраструктурою утворюється трьохрівнева модель взаємодії, де запит спочатку надходить до сервера автоматизації, після цього до одного з кластерів, і потім до окремого вузла в кластері, який запускає ефемерний контейнер з кроками процесу автоматизації.

Взаємодія між сервером автоматизації і кластером при цьому є односторонньою, де ініціатором завжди виступає сервер. Хмарні обчислювальні ресурси використовуються лише для виконання кроків процесу автоматизації. Обробка логів, метрик, чи моніторинг стану задачі виконується основним вузлом сервера автоматизації. Означену модель взаємодії представлено схематично на рисунку 1.

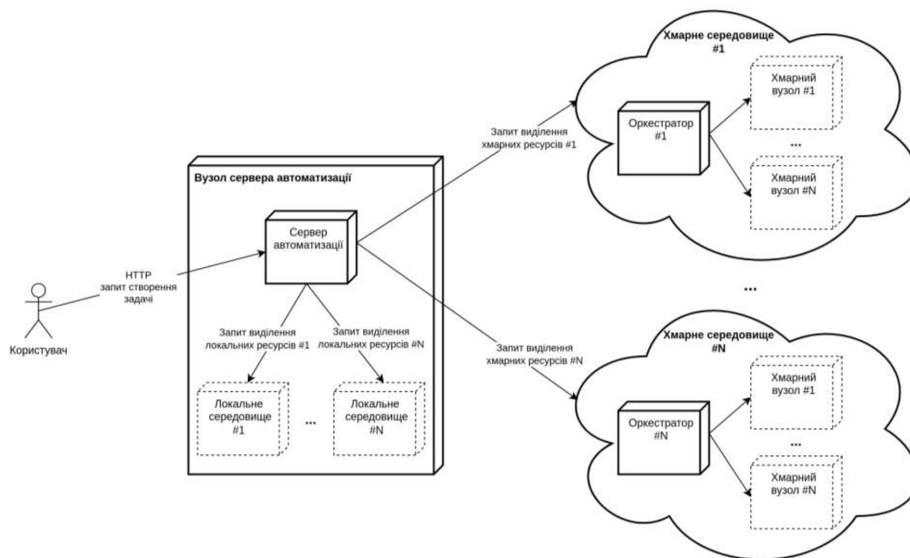

Рисунок 1. Модель взаємодії сервера автоматизації з хмарою

Архітектура сервера автоматизації

Розроблений сервер автоматизації використовує програмні задачі для опису процесів автоматизації. Сервер надає засоби для опису, розподілу між середовищами виконання та моніторингу програмних задач, тобто виконує оркестрацію задач.

Архітектура розробленого сервера базується на шаблоні “Плагін” [12], в межах якого виділяються два основних елементи – статичне ядро та динамічні плагіни. Кількість плагінів є чітко визначеною, кожен з них є обов’язковим і вирішує одну з функціональних задач сервера автоматизації. До таких задач можна віднести інтеграцію з середовищем виконання, контроль ходу і розкладу виконання задач, обробку логів і метрик, управління конфігурацією тощо. Плагіни є незалежними, а тому пряма взаємодія між ними не відбувається. Композиція викликів плагінів реалізована на рівні ядра серверу, яке залежить від абстрактних інтерфейсів плагінів, а не їхніх конкретних реалізацій. Кожен з плагінів може мати декілька реалізацій. Користувачу надається можливість створити власні реалізації плагінів за допомогою JavaScript-сумісної мови програмування, за рахунок чого досягається можливість адаптації сервера відповідно до вимог користувача. Повний перелік функціональних задач плагінів наведено у таблиці 1.

Таблиця 1. **Перелік плагінів**

Плагін	**Функціональні задачі**
envBuilder	Підготовка/видалення шаблону середовища
envRunner	Запуск задачі в середовищі на основі шаблону; отримання потоків логів і метрик задачі; видалення/зупинка задачі; моніторинг стану задачі
config	Управління конфігурацією
log	Обробка і пошук логів
metric	Обробка і пошук метрик
queue	Управління порядком виконання задач
scheduler	Управління розкладом виконання задач

Усі плагіни, крім резолвера конфігурації (config), використовуються у ході підготовки чи безпосередньо виконання процесу автоматизації. Резолвер використовується лише на етапі ініціалізації сервера автоматизації для передачі параметрів конфігурації іншим плагінам.

Найпершим у життєвому циклі задачі є будівельник середовища (envBuilder), що готує шаблон середовища на етапі створення задачі. Після цього планувальник (scheduler) реєструє відкладений процес, що розпочне виконання задачі відповідно до тригера. Після того, як планувальник вказує, що настав час виконання, відповідна задача додається до черги (queue), де очікує, поки не з’явиться вільний обчислювальний ресурс. Наступним кроком є виконання задачі. Для цього

ISSN 1560-8956

використовується плагін виконувача середовища (envRunner), що запускає задачу у межах екземпляру середовища, що створюється на основі раніше підготовленого шаблону. Якщо створення середовища пройшло успішно, то плагін повертає об’єкт, за допомогою якого сервер автоматизації може взаємодіяти з цим середовищем – зупиняти чи видаляти його, отримувати логи чи метрики, очікувати завершення виконання процесу автоматизації в ньому тощо. Далі відбувається моніторинг стану задачі, у ході якого сервер автоматизації отримує потоки логів і метрик, що передаються для обробки плагінам логів (log) і метрик (metric) відповідно. Після завершення виконання задачі сервер знищує екземпляр середовища, очікує завершення обробки логів і метрик, а також зберігає всю потрібну інформацію про поточне виконання до БД. Якщо поточне виконання задачі є останнім, то додатково сервер автоматизації зупиняє її наступні заплановані виконання за допомогою планувальника (scheduler), а також видаляє шаблон середовища засобами будівельника середовища (envBuilder).

Зауважимо, що взаємодія з середовищем виконання задач відбувається лише на рівні плагінів envBuilder й envRunner, а тому реалізація інтеграції з будь-яким іншим середовищем потребує створення реалізацій плагінів, розроблених користувачем. Внесення змін до ядра сервера автоматизації чи інших плагінів при цьому не потрібне. Перебіг взаємодії між плагінами зображено схематично на рисунку 2.

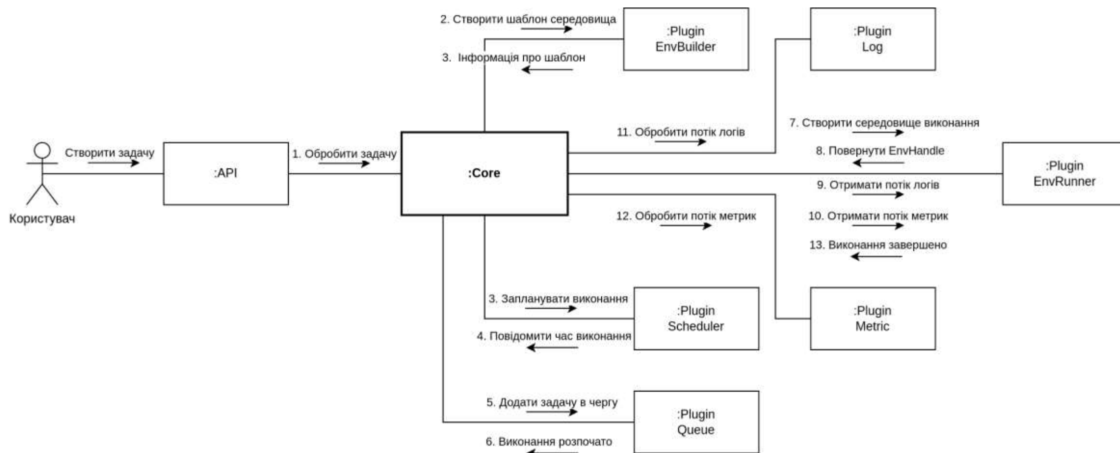

Рисунок 2. Схема взаємодії між плагінами і ядром сервера автоматизації

Інтеграція з Kubernetes

Інтеграція використовує лише нативні засоби Kubernetes, а саме – його REST API [11]. За таких умов єдиною вимогою до кластера є доступність його REST API для сервера автоматизації. Налаштування на рівні сервера при цьому зводиться до вказування параметрів підключення до кластера. Такими параметрами є адреса кластера, а також токен відповідного Service Account.

Kubernetes, як і будь-яке інше середовище виконання задач, повинен реалізувати створення шаблонів й екземплярів середовища. Інакше кажучи, виділяти обчислювальні ресурси для виконання кроків процесу автоматизації. Оскільки Kubernetes реалізує інтеграцію з containerd [11], то шаблон середовища можна представити у вигляді Docker Image [13]. На основі шаблону створюється екземпляр середовища, що представлено у вигляді найменшої одиниці обчислювальних ресурсів в кластері – Kubernetes Pod.

При описі програмної задачі користувач повинен вказати базовий Docker Image, на основі якого буде створено середовище виконання задачі. Надалі кожне виконання задачі призводить до створення Pod в межах кластеру, що утворюється на основі раніше вказаного Docker Image. Створений Pod вміщує контейнер, вхідним скриптом якого (CMD) є кроки процесу автоматизації. Важливо зауважити, що CMD скрипт передається саме у момент створення Pod, а не шаблону середовища. Тобто, проміжний Docker Image не створюється, а підготовка шаблону зводиться лише до перевірки валідності вказаного Docker Image. Це є необхідним, зважаючи на розподілену природу Kubernetes. Сервер автоматизації є часто фізично ізольованим від кластера, а тому створення проміжних Docker Image вимагає використання зовнішніх Docker реєстрів. Натомість передача CMD скрипта у момент створення Pod дозволяє цього уникнути.

На відміну від деяких аналогів [1], сервер автоматизації не створює додаткові контейнери для побудови комунікації з кластером. Натомість у межах кластера виконуються лише кроки процесу автоматизації, що були вказані користувачем при створенні задачі. Всі інші дії, пов'язані з моніторингом задачі чи обробкою логів і метрик, виконуються на основному вузлі сервера автоматизації.

Kubernetes реалізує асинхронний підхід щодо створення ресурсів. Іншими словами, у більшості випадків виклик ендпоінту для створення Pod повертає успішний результат [11]. При цьому власне виділення вузла, створення контейнера, виконання кроків процесу автоматизації відбуваються у фоновому процесі, жодна інформація про який не повертається в межах відповіді до HTTP запиту створення Pod. Важливо розуміти, що на кожному із зазначених вище етапів може виникнути помилка. Зважаючи на це, сервер автоматизації відстежує будь-які зміни стану Pod. Це досягається за допомогою Kubernetes Watch. Цей засіб створює Server Sent Events (SSE) канал між сервером автоматизації та кластером, по якому Kubernetes надсилає події про зміни стану вказаного Pod [11]. Однак у межах дослідження більш важливим є стан внутрішнього контейнера, що міститься в межах цього Pod. Всього виділяють три таких стани: Waiting, Running, Terminated.

Контейнер перебуває у Waiting стані, якщо його все ще не було запущено, тобто виконання скрипту в CMD не було розпочато. На цьому етапі для Pod виділяється робочий

вузол, відбувається спроба завантаження шаблону середовища тощо. Якщо помилка виникає на цьому етапі, то життєвий цикл задачі можна вважати завершеним, адже оскільки контейнер не було створено, то збір логів чи метрик не є доцільним.

Якщо ж контейнер перейшов у стан Running, то середовище було успішно створено і розпочалося виконання кроків процесу автоматизації. Саме при переході у цей стан доцільно розпочати збір логів і метрик. Збір повинен продовжуватися, доки контейнер не перейде у Terminated стан.

Контейнер переходить у стан Terminated незалежно від результату виконання процесу автоматизації. Важливо зауважити, що інколи контейнер може завершити виконання настільки швидко, що час його перебування у стані Running є близьким до нуля. Власне тому сервер автоматизації завжди виконує щонайменше одну ітерацію збору логів та метрик незалежно від того, чи є контейнер активним у цей проміжок часу. Варто зауважити, що перехід у Terminated стан може бути ініційованим користувачем. Наприклад, якщо програмну задачу було зупинено, чи видалено, то контейнер отримує SIGTERM чи SIGKILL сигнали відповідно, що призводять до завершення його виконання.

Kubernetes API надає нативний ендпоінт для отримання логів Pod, що дозволяє утворити довготривалий HTTP канал, по якому Kubernetes надсилає логи по мірі їхнього виникнення [11]. Таким чином, отримання і обробка логів є реактивними, а запит до Kubernetes API виконується всього один. Оскільки логи є потоковими даними, що не гарантують однорідності надходження у часі чи об'ємі, то для вирішення класичної проблеми "producer-consumer" у контексті обробки логів, використовуються потоки (Readable Stream) в Node.js [14]. Для того, аби перетворити логи з текстового формату в стандартизований вигляд сервера автоматизації, використовуються потоки трансформації (Transform Stream).

Kubernetes не надає нативного рішення для отримання метрик засобами REST API, на відміну від логів [11]. Зважаючи на те, що розроблене рішення уникає внесення змін до кластеру, то за замовчуванням збір метрик не відбувається. Водночас пропонується рішення, що дозволяє їх збір за допомогою інструментів, які не є нативними – Metrics Server і Prometheus.

Metrics Server є розширенням Kubernetes кластеру, що дозволяє отримувати інформацію про використання CPU чи RAM вузлами чи Pod за допомогою команди kubectl top, або ж ендпоінту /metrics [15]. Kubernetes не надає можливості отримувати метрики реактивно в межах єдиного HTTP каналу. Тому сервер автоматизації повинен регулярно виконувати запити і перевіряти, чи сталося оновлення відповідних метрик. Таким чином, якщо оновлення метрик сталося, то відбувається їх приведення до стандартизованого формату і додавання до потоку (Readable Stream), що надалі буде оброблено плагіном метрик [14]. Варто зауважити, що Metrics Server не гарантує

абсолютної точності чи високої інтенсивності заміру метрик (заміри стаються кожні 15-60 секунд), а тому не є рекомендованим для виконання замірів у режимі реального часу [15]. Проте, зважаючи на популярність цього інструменту, а також того, що навіть таких метрик може бути достатньо для виявлення аномалій виконання задач, інтеграцію з Metrics Server включено для сервера автоматизації.

На відміну від Metrics Server, Prometheus було спроєктовано для моніторингу контейнерів в режимі реального часу в розподілених середовищах [16]. Prometheus при цьому не є частиною Kubernetes і може бути розгорнутий поза його межами, а тому інтеграція з цим інструментом вимагає надання адреси розташування сервера Prometheus. Як і у випадку з Metrics Server, реактивне отримання метрик не є можливим [16], а тому сервер автоматизації виконує регулярні запити для отримання метрик за допомогою мови PromQL. Зауважимо, що Prometheus є лише інструментом для збору і отримання метрик [16]. Генерація метрик виконується на рівні Kubernetes кластеру. Одним із розширень для генерації таких метрик є kube-state-metrics. Власне, саме ці метрики і використовує сервер автоматизації. Архітектуру означеної інтеграції з Kubernetes зображено схематично на рисунку 3.

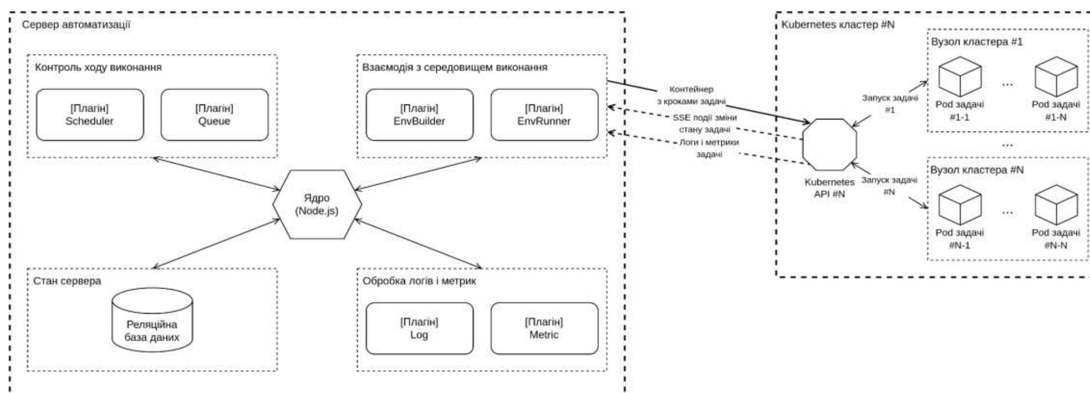

Рисунок 3. Архітектура інтеграції з Kubernetes

Аналіз запропонованого рішення

Оскільки спрощення складно виразити кількісно, то для порівняння розробленої архітектури з існуючими сформульовані такі якісні характеристики архітектури:

- використання односторонньої моделі взаємодії з хмарою;
- відсутність потреби встановлення агентного ПЗ в хмарі;
- виконання в межах хмари лише кроків процесу автоматизації;

Одностороння модель взаємодії є концептуально простішою, а тому дозволяє спростити налаштування мережевої взаємодії. Водночас, така модель збільшує затримки в оновленні стану кластера.

 ISSN 1560-8956

Використання нативних засобів кластерів замість агентного ПЗ значно спрощує налаштування інтеграції сервера з хмарою. Проте унеможливлює пряму взаємодію з окремими вузлами і надає менше контролю над хмарою.

Перенесення обробки логів і метрик до вузла сервера автоматизації дозволяє розмежувати виконання кроків процесу автоматизації і обробку проміжних результатів його роботи. Це дозволяє зменшити навантаження на кластер, однак збільшує навантаження на вузол сервера автоматизації.

Вимоги як до сервера автоматизації, так і до хмарної інфраструктури залежать від потреб користувача. Зважаючи на це, при аналізі важливо врахувати різні сценарії використання розробленого рішення. У дослідженні розглядаються три сценарії, що залежать від внутрішнього представлення хмари. Опис сценаріїв наведено у таблиці 2.

Таблиця 2. **Сценарії використання**

<table>
<tr><th>Сценарій</th><th>Аудиторія</th><th>Запропоноване рішення</th><th>Аналоги</th></tr>
<tr><td>Один вузол</td><td>Об’єм задач є незначним і рівномірним</td><td rowspan="3">Необхідно надати серверу параметри підключення до кластера. Зміни до хмари не вносяться</td><td>На вузлі необхідно встановити агентне ПЗ</td></tr>
<tr><td>Один кластер</td><td>Об’єм задач є надмірним для одного вузла</td><td rowspan="2">На рівні кластера (GitHub Actions) або окремого вузла в кластері (GitLab CI/CD, BitBucket Pipelines, Jenkins) необхідно динамічно (Jenkins) чи попередньо (інші аналоги) встановити агентне ПЗ</td></tr>
<tr><td>Кілька кластерів</td><td>Об’єм задач є надмірним для одного кластера або наявні вимоги до вендора чи географічного розташування хмари</td></tr>
</table>

Запропоноване рішення не вносить жодних змін до хмарної інфраструктури для будь-якого зі сценаріїв, адже використовує лише нативні засоби оркестраторів контейнерів. Розроблений сервер автоматизації ускладнює налаштування для сценарію з одним кластером, адже не реалізує пряму взаємодію з окремими вузлами, а тому вимагає навіть у такому випадку створювати кластер.

Водночас, у контексті сценарію з одним чи кількома кластерами спрощує налаштування, адже зводить конфігурацію лише до вказання параметрів підключення до кластера, не вимагаючи при цьому встановлення агентного ПЗ, як це роблять більшість аналогів. Схожий підхід реалізує й Jenkins, проте встановлює таке ПЗ динамічно, що вносить тимчасові зміни до хмарної інфраструктури й використовує додаткові обчислювальні ресурси.

Висновки

Отже, результатом дослідження є розроблена архітектура сервера автоматизації, що реалізує програмні засоби для розгортання процесів автоматизації у вигляді програмних задач у хмарних чи локальних середовищах з можливістю подальшого

моніторингу. Сервер реалізує односторонню модель взаємодії з хмарою, що досягається нативними засобами оркестраторів контейнерів, тому не потребує встановлення агентного ПЗ в межах хмарної інфраструктури. Архітектура включає плагіни, кожен з яких вирішує одну з функціональних задач сервера автоматизації і може бути заміненим на реалізацію користувача, що дозволяє адаптувати сервер до різних вимог.

Спроєктована архітектура може бути використана для більшості оркестраторів контейнерів, зокрема, у дослідженні виконана реалізація для Kubernetes. Сервер використовує Kubernetes API для створення Pod для кожного виконання програмної задачі. Моніторинг стану задачі досягається за допомогою Kubernetes Watch, збір логів – засобами нативного ендпоінту, отримання метрик – за допомогою Metrics Server або Prometheus і kube-state-metrics.

Запропоноване рішення спрощує розгортання процесів автоматизації для користувачів, що використовують один чи кілька кластерів у хмарі, адже конфігурація сервера відбувається вказуванням параметрів підключення до кластера і не потребує встановлення агентного ПЗ. Водночас використання запропонованого підходу для окремих вузлів привносить додаткову складність, адже не реалізує пряму взаємодію з хмарними одиницями і вимагає розгортання кластера у такому випадку.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Jenkins User Documentation. URL: https://www.jenkins.io/doc/
2. GitHub Actions Documentation. URL: https://docs.github.com/en/actions
3. Use CI/CD to build your application – GitLab. URL: https://docs.gitlab.com/ee/ci/
4. Bitbucket Cloud Resources. URL: https://support.atlassian.com/bitbucket-cloud/resources/
5. Fowler M., Foemmel M. Continuous integration. 2006
6. A. Kanitz et al. The GA4GH Task Execution API: Enabling Easy Multi Cloud Task Execution. 2024
7. Malawski M. Towards Serverless Execution of Scientific Workflows – HyperFlow Case Study. WORKS 2016 Workshop. 2016
8. H. Zhao et al. Supporting Multi-Cloud in Serverless Computing 2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing
9. Koskela L. Platform independent job workload management: Master of Science Thesis. 2020. URL: https://trepo.tuni.fi/bitstream/handle/10024/123555/KoskelaLauri.pdf
10. T. Kiss et al. A cloud-agnostic queuing system to support the implementation of deadline-based application execution policies Elsevier. 2019
11. Kubernetes Documentation: https://kubernetes.io/docs/home

12. Fowler M. Plugin. Patterns of enterprise application architecture: pattern Enterpr Applica Arch. 2012. P. 499

13. Docker Docs. URL: https://docs.docker.com/

14. Node.js documentation. URL: https://nodejs.org/api/stream.html

15. Kubernetes Metrics Server. URL: https://kubernetes-sigs.github.io/15/

16. Overview – Prometheus. URL: https://prometheus.io/docs/introduction/overview/