DOI: 10.20998/2079-0023.2024.02.15
УДК 004.9

Н. Є. ХАЦЬКО, кандидат технічних, доцент, доцент кафедри інформаційних систем та технологій Національного технічного університету «Харківський політехнічний інститут», Харків, Україна; e-mail: nataliia.khatsko@khpi.edu.ua; ORCID: https://orcid.org/0000-0002-2543-0280
М. В. СЛЄПУШКОВ, студент магістратури, Національний технічний університет «Харківський політехнічний інститут», м. Харків, Україна; e-mail: m.sliepushkov@gmail.com; ORCID: https://orcid.org/0009-0001-0004-2820
К. О. ХАЦЬКО, старший викладач кафедри інформаційних систем та технологій Національного технічного університету «Харківський політехнічний інститут», аспірант, Харків, Україна; e-mail: kyrylo.khatsko@khpi.edu.ua; ORCID: https://orcid.org/0000-0003-3315-1553
Є. О. ШЕБАНОВ, аспірант, Національний технічний університет «Харківський політехнічний інститут», Харків, Україна; e-mail: yevhenii.shebanov@cs.khpi.edu.ua; ORCID: https://orcid.org/0009-0006-9032-8764

МОДИФІКОВАНИЙ АЛГОРИТМ РОЗГОРТАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ З ВИКОРИСТАННЯМ БАГАТОПОТОЧНОСТІ

У статті представлено модифікований алгоритм розгортання програмного забезпечення з використанням багатопоточності в AWS CodeBuild, спрямований на оптимізацію часу збірки та зниження витрат на обчислювальні ресурси в хмарному середовищі. На основі скінченних автоматів, часових автоматів та мереж Петрі було змодельовано основні етапи процесу збірки, включаючи паралельне виконання тестів, аналіз розподілу завдань та управління обчислювальними ресурсами. Особлива увага приділялася ідентифікації та усуненню обмежень стандартних механізмів паралелізації AWS CodeBuild, які можуть призводити до неефективного використання ресурсів та збільшення тривалості збірки. Дослідження виявило, що стандартні механізми AWS CodeBuild не завжди здатні оптимально використовувати системні ресурси, особливо при обробці великих програмних проектів із численними залежностями. Для подолання цих обмежень було запропоновано використання багатопоточності засобами Python, як зручного інструменту для розширення базового функціоналу. Запропонований підхід дозволив реалізувати гнучке керування потоками та розподіл завдань на рівні користувацьких сценаріїв, зменшивши загальний час збірки. Експериментальні результати показали значне скорочення часу виконання збірок у порівнянні зі стандартними налаштуваннями AWS CodeBuild. Це підтверджує ефективність використання запропонованого алгоритму для збільшення продуктивності та забезпечення високої масштабованості процесів збірки в хмарному середовищі. Розроблений алгоритм є особливо актуальним для великих програмних проектів, які вимагають частих ітераційних збірок та тестування. Отримані результати можуть бути використані для вдосконалення автоматизованих процесів розгортання та управління обчислювальними ресурсами у хмарних середовищах.

Ключові слова: процес розгортання, інформаційна технологія, алгоритм, модель процесу розгортання програмного забезпечення, скінченний автомат, часовий автомат, мережі Петрі.

Вступ. У сучасному програмному середовищі, особливо у хмарних інфраструктурах, таких як AWS CodeBuild, ефективність збирання програмного забезпечення є важливим фактором, який впливає як на швидкість розгортання, так і на вартість використання ресурсів. Традиційний послідовний підхід до виконання збірки та тестування має обмежену продуктивність через необхідність послідовного виконання окремих етапів, що може спричиняти тривалі затримки, особливо при роботі з великими проєктами. Оскільки CodeBuild оплачується відповідно до часу використання, оптимізація цього часу є першочерговою для зменшення витрат. Одним із способів вирішення цієї проблеми є впровадження багатопоточності, що дозволяє паралельно виконувати незалежні етапи. Водночас, недостатнє теоретичне обґрунтування механізмів паралельного виконання ускладнює оптимальне впровадження таких підходів. Необхідно розробити модель, яка враховує специфіку процесів збирання та дозволяє визначити оптимальні умови для використання паралельних потоків. Для постановка цієї проблеми доцільно застосувати формалізми такі як: скінченні автомати та мережі Петрі, що дозволяє описати процес збірки та змоделювати використання багатопоточності.

Мета дослідження полягає в розробці модифікованого алгоритму розгортання, що допоможе зменшити вартість використання хмарних ресурсів. План дослідження складається з наступних етапів: аналіз традиційного алгоритму розгортання, модифікація алгоритму, приведення експерименту в якому вимірюються характеристики обох процесів розгортання, порівняння характеристик базового алгоритму розгортання та модифікованого.

У рамках дослідження було проаналізовано низку наукових робіт, що розкривають сучасні підходи до використання теорії автоматів у різних аспектах розробки програмних систем та їх оптимізації. У роботі [1] розглянуто застосування псевдопам’яті скінченних автоматів для створення криптографічних систем. Псевдопам’ять забезпечує швидкодію та гнучкість у шифруванні даних, що є перспективним для безпечного управління хмарними системами. Дослідження [2] пропонує ефективні методи індексації та компресії скінченних автоматів, які можуть бути використані для оптимізації зберігання та пошуку даних у великих хмарних середовищах, що підвищує ефективність обробки регулярних запитів. Інструмент ComVis, представлений у [3], є прикладом візуалізації автоматів, орієнтованої на освітні цілі, але підхід до інтерактивного моделювання може бути адаптований для управління складними процесами розгортання в хмарі [4]. У [5] запропоновано концепцію паралельних скінченних автоматів (parallel finite automata – PFAs), які підтримують моделювання багатопоточних систем із точним

контролем станів. Це є актуальним для оптимізації паралельних потоків у процесі зборки програмного забезпечення. Автори демонструють, як PFAs допомагають уникати нескінченних просторів станів, притаманних деяким мережам Петрі, зберігаючи при цьому можливість моделювати паралельні процеси. У роботі показано, що PFAs, хоч і еквівалентні детермінованим скінченним автоматам за мовою, але структурно простіші. Таким чином PFAs кращі для моделювання багатопоточності.

Дослідження [6] вводить новий клас мереж Петрі – Gadara, які використовуються для уникнення взаємоблокувань у багатопотокових середовищах, що є особливо важливим для забезпечення стабільності хмарних систем. Робота [7] демонструє підходи до перевірки моделей багатопотокових програм за допомогою асинхронних методів, які забезпечують стабільну роботу паралельних процесів і знижують ризик збоїв у системах. Інструмент Ballet, описаний у [8], автоматизує координацію децентралізованої переконфігурації в хмарних середовищах, дозволяючи значно знизити залежність від централізованого управління та збільшити швидкість і надійність процесів. У [9] описано ефективність використання фреймворка для обробки автоматів у хмарних середовищах, що забезпечує високу пропускну здатність і гнучкість конфігурації для складних систем. Особливо актуально це для розгортання додатків, які мають мікросервісну архітектуру [10]. Дослідження [11] підкреслює можливість застосування кінцевих автоматів для аналізу питань довіри у хмарних сервісах, що дозволяє структурувати управління станами та забезпечувати безпеку даних. У роботі [12] представлено інструмент для симуляції та тестування детермінованих скінченних автоматів, що дозволяє спростити відлагодження і перевірку програм, побудованих на основі автоматів. Такий інструмент міг бути корисним для дослідження, що проводиться, але не застосований через відсутність підтримки симуляції багатопоточності.

В статті [13] вперше була описана концепція часових автоматів – це різновид скінченних автоматів, який включає додаткові змінні, відомі як годинники, що дозволяють враховувати час між переходами між станами. На відміну від традиційних скінченних автоматів, де перехід відбувається миттєво при спостереженні певних подій або умов, часові автомати дозволяють моделювати системи, в яких переходи залежать від проміжку часу, що минув з моменту останнього переходу або події. Це робить їх особливо корисними для опису і верифікації систем з реальним часом, таких як вбудовані системи, програмне забезпечення з часовими обмеженнями тощо. Система розгортання, що досліджується, має часові обмеження, тому часові автомати доцільно використати.

Таким чином, в дослідженні скористаємось концепцією часових скінченних автоматів та мережами Петрі для уникнення взаємоблокувань у багатопотокових середовищах, що відповідає нашій меті теоретичного моделювання паралельних потоків на одному з етапів збірки.

Огляд наукових статей та результати попередніх досліджень різних авторів підтверджує актуальність використання теорії автоматів для розв'язання задач моніторингу та оптимізації процесів розгортання у хмарному середовищі є актуальним.

Методи дослідження. Теорія автоматів представляє собою математичний апарат, який дозволяє описувати та аналізувати поведінку дискретних систем. У контексті розгортання програмних систем прийнятні скінченні автомати та їх розширення. Часовий автомат – це розширення класичного скінченного автомата, яке дозволяє враховувати часові обмеження для переходів між станами або перебування в певних станах. Він широко використовується для моделювання систем реального часу, де час є важливим фактором (наприклад, керування транспортними системами, аналіз протоколів зв'язку, хмарні обчислення). Часовий автомат розширює класичний автомат, додаючи додаткові поняття. Деякі з них:

- годинники – набір змінних, які завжди збільшуються зі швидкістю 1 (реальне значення часу). Їх можна обнуляти при виконанні певних дій;
- стан – описує поточний стан системи з додаванням часових умов;
- переходи – вказують реакцію автомату на впливи, мають часові умови (гвардії), які визначають, коли перехід дозволено;
- часові обмеження – умови, накладені на годинники. Окрім гвардій, які повинні виконуватися для здійснення переходу, існують інваріанти, що визначають час знаходження системи в певному стані.

Використання часових автоматів при формалізації процесу збирання програмного забезпечення за допомогою AWS CodeBuild є абсолютно доречним та корисним, оскільки вони дозволяють точно моделювати часові обмеження на кожному етапі збірки, враховуючи послідовність виконання завдань і можливість паралельного виконання. Завдяки цим моделям можна формалізувати часові характеристики етапів, таких як завантаження вихідного коду, встановлення залежностей, виконання збірки та тестів.

AWS CodeBuild має чітко визначені етапи збірки, які в автоматі представлені як стани. Час кожного етапу вимірюється, крім того ми маємо обмеження на загальний час збірки. В разі перевищення цього обмеження збірку буде завершено.

Опишемо етапи збірки (стани автомату).

1. SUBMITTED – задачу збірки створено, і вона очікує на запуск у черзі.
2. QUEUED – задача перебуває в черзі на виконання.
3. PROVISIONING – CodeBuild налаштовує середовище збірки, включаючи створення обчислювальних ресурсів.
4. DOWNLOAD_SOURCE – завантаження вихідного коду з визначеного джерела.
5. INSTALL – встановлення необхідних залежностей та інструментів, визначених у файлі конфігурації збірки (buildspec.yml).

6. PRE_BUILD – підготовка до основного процесу збірки, наприклад, виконання скриптів ініціалізації чи підготовчих дій.

7. BUILD – основний етап збірки, на якому виконуються всі завдання, необхідні для створення продукту (компіляція, збирання, тестування, генерація результатів).

8. POST_BUILD – завершальні дії, упаковка артефактів, створення Docker-образів тощо.

9. UPLOAD_ARTIFACTS – завантаження результатів збірки у місце зберігання, наприклад, Amazon S3, Docker репозиторій.

10. ERROR – процес завершився з помилкою; збірку не вдалося виконати.

11. FINALIZING – завершення процесу, очищення середовища збірки, підготовка звітів.

12. COMPLETED – завдання завершено.

Функція переходів у теорії часових автоматів визначає, як і за яких умов відбувається перехід між станами. В контексті нашого завдання вона може бути записана як:

$$\delta : L \times B(C) \to 2^L,$$

де L – множина станів, що описані вище; $B(C)$ – булеві вирази (гвардії), що задають умови переходу на основі відліку часу; 2^L – множина підмножин станів множини L, куди можливий перехід.

Визначимо функцію переходів δ.

1. Початковий стан:

$$\delta(\text{SUBMITTED}, x \le t_1) = \{\text{QUEUED}\}.$$

2. Перехід між етапами: Для кожного стану визначається наступний стан та часова гвардія $x \le t_1$, яка гарантує, що перехід відбувається лише в допустимий проміжок часу:

$$\delta(\text{SUBMITTED}, x \le t_2) = \{\text{PROVISIONING}\},$$

$$\delta(\text{PROVISIONING}, x \le t_3) = \{\text{DOWNLOAD\_SOURCE}\},$$

$$\delta(\text{DOWNLOAD\_SOURCE}, x \le t_4) = \{\text{INSTALL}\}.$$

І так далі, до завершального етапу.

3. Перехід у стан ERROR – якщо годинник перевищує допустимий час t_i, автомат переходить до стану ERROR:

$$\delta(l, x > t_i) = \{\text{ERROR}\},$$

де l – один з перших дев'яти станів, що описані вище та на рис. 1.

4. Завершення роботи – зі стану FINALIZING автомат завжди переходить до COMPLETED:

$$\delta(\text{FINALIZING, true}) = \{\text{COMPLETED}\}.$$

Таким чином, функцію переходів визначаємо як:

$$\delta(l, g) = \begin{cases} \{l'\}, & \text{якщо } g = (x \le t_i), \\ \{\text{ERROR}\}, & \text{якщо } g = (x > t_i), \\ \{\text{COMPLETED}\}, & \text{якщо } g = \text{true}, \end{cases} \quad (1)$$

де l' – наступний стан; $g = (x \le t_i)$ – гвардія для кожного переходу, визначає максимальний час t_i, допустимий для поточного стану; ERROR – стан, що позначає виникнення помилки, якщо час у стані перевищує допустимий t_i.

Функція переходів δ дозволяє чітко описати логіку переходів у часовому автоматі збірки програмного забезпечення (табл. 1).

Графічно автомат виглядає як орієнтований граф, де кожен стан представлений вузлом, а переходи між станами - стрілками (див. рис. 1).

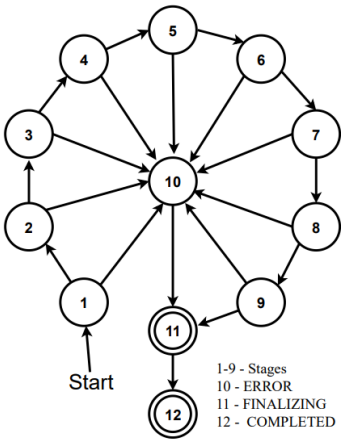

Рис. 1. Скінченний автомат збірки програмного забезпечення в AWS CodeBuild

Таблиця 1 – Таблиця визначення функції переходів

Номер	Поточний стан (l)	Наступний стан (l')	Гвардія (g)	Дія (Σ)
1	SUBMITTED	QUEUED	$x \le t_1$	Завдання додано до черги.
2	QUEUED	PROVISIONING	$x \le t_2$	Початок налаштування середовища.
3	PROVISIONING	DOWNLOAD_SOURCE	$x \le t_3$	Завантаження джерел коду.
4	DOWNLOAD_SOURCE	INSTALL	$x \le t_4$	Інсталяція залежностей.
5	INSTALL	PRE_BUILD	$x \le t_5$	Підготовчі дії перед збіркою.
6	PRE_BUILD	BUILD	$x \le t_6$	Початок основного процесу збірки.
7	BUILD	POST_BUILD	$x \le t_7$	Завершення збірки.
8	POST_BUILD	UPLOAD_ARTIFACTS	$x \le t_8$	Завантаження артефактів у сховище.
9	UPLOAD_ARTIFACTS	FINALIZING	$x \le t_9$	Завершення роботи.
11	FINALIZING	COMPLETED	$x \le t_{10}$	Завдання завершено успішно.
	ANY_STATE	ERROR	$x > t_{max}$	Перехід до обробки помилки.
10	ERROR	FINALIZING	Завжди	Очищення та завершення роботи.

У процесі збирання програмного забезпечення кожен етап зазвичай слідує за попереднім, але існують завдання, які можуть виконуватися незалежно та паралельно. Це дозволяє ефективніше використовувати доступні ресурси і скорочувати загальний час збірки. Наприклад, завантаження залежностей і виконання юніт-тестів є завданнями, які часто не залежать один від одного і можуть виконуватися одночасно. Так, всередині 7-го стану (BUILD), як правило, крім збірки виконуються ще й юніт-тести. Для моделювання їх паралельного виконання ми застосуємо мережі Петрі, які добре підходять для формалізації систем, що включають одночасні, залежні та незалежні процеси.

При побудові моделі тестування за допомогою мереж Петрі кожен тест можна уявити як окремий процес, представлений місцем у мережі. Переходи між цими місцями позначають виконання тестів, а загальний результат тестування стає доступним лише після завершення всіх окремих процесів. Це дозволяє побудувати схему, яка відображає як паралельне виконання тестів, так і їх об'єднання в єдиний завершений результат.

Мережу Петрі описують як кортеж:

$$PN = (P, T, f, M),$$

де $P = \{p_0, p_1, p_2, \ldots, p_n, p_{\text{done}}\}$ – множина місць. p_0 відповідає початку процесу, p_1, p_2, …, p_n – місця для виконання окремих тестів, p_{done} – місце завершення тестування;

$T = \{t_1, t_2, \ldots, t_n, t_{\text{finish}}\}$ – множина переходів, t_1, t_2, …, t_n відповідають запуску і завершенню кожного тесту, t_{finish} відповідає завершенню тестування після виконання всіх тестів;

F – множина дуг, яка задає відношення між місцями і переходами; з рис. 2 виходить, що

$$F = \{(p_0, t_1), (t_1, p_1), \ldots, (p_0, t_n), (t_n, p_n), (p_1, t_{\text{finish}}), \ldots, (p_n, t_{\text{finish}}), (t_{\text{finish}}, p_{\text{done}})\}.$$

M – множина важелів дуг. В контексті цієї моделі це стале значення, що не має впливу.

Паралельне виконання тестів змодельовано та представлено на рис. 2. Мережа Петрі моделює процес паралельного виконання тестування в системі збирання програмного забезпечення.

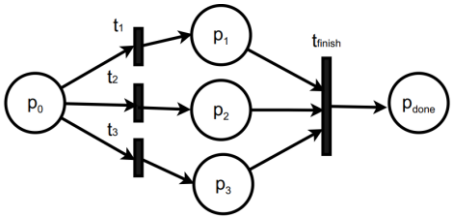

Рис. 2. Схема паралельних процесів з використанням Мережі Петрі

На рис. 2 модель містить описані нижче елементи.

1. Місця:
   - p_0 – початковий стан, тести ще не запущені;
   - p_1, p_2, p_3 – місця (стани), які відповідають активним станам виконання окремих тестів;
   - p_{done} – кінцевий стан, всі тести завершені.
2. Переходи:
   - t_1, t_2, t_3 – переходи, які запускають виконання кожного з тестів;
   - t_{finish} – перехід після завершення всіх тестів.

Логіка роботи мережі Петрі в контексті паралельного тестування наступна.

1. В стані системи p_0 всі тести завантажені.
2. Переходи t_1, t_2, t_3 активуються одночасно, переміщуючи виконання тестів до станів p_1, p_2, p_3.
3. Після завершення тестів активується перехід t_{finish}, який переміщує тести до стану p_{done}, що сигналізує завершення тестування.

Формула часу для послідовного виконання тестів:

$$T_{\text{посл}} = T_{\text{синх}} + \sum_{i=1}^{n} T_i + T_{\text{заверш}},$$

де $T_{\text{синх}}$ – час синхронізації перед запуском тестування; T – час виконання i-го тесту; $T_{\text{заверш}}$ – час завершального етапу тестування.

Формула для часу для паралельної моделі:

$$T_{\text{пар}} = t_{\text{синх}} + \max(t_1, t_2, \ldots, t_n) + t_{\text{заверш}}, \qquad (2)$$

де $t_{\text{синх}}$ – час синхронізації на початку тестування, що необхідний для розподілу задач між потоками; t_1, t_2, …, t_n – час виконання кожного з паралельних тестів; $t_{\text{заверш}}$ – час синхронізації системи після завершення тестів.

Паралельна природа процесу означає, що загальний час визначається найдовшим з усіх паралельних тестів.

Експеримент. Для перевірки теоретичних припущень було створено проєкт на AWS CodeBuild та написано програму, до якої застосовувалися тести.

Коректність моделі роботи системи (рис. 1, 2) була перевірена шляхом створення помилок на різних етапах збірки продукту. А саме на рис. 3 та рис. 4 зображені стани, що завершились помилками, після яких збірка не продовжувалась, а процес переходив до стану FINALIZING та COMPLITED. Таким чином, зв'язки попередніх станів зі станом ERROR були підтверджені. На рис. 5–7 продемонстровано успішне виконання, коли пройдено всі етапи збірки без помилок. Кожен етап має свій час виконання, а також обмеження цього часу, в разі перевищення ліміту часу збірка перепиває ться. Що і враховано в (1), (2).

Окремого етапу для тестування в CodeBuild не передбачено тому тести були виконані на етапі BUILD.

Спочатку було виконано тестування в однопоточному. режимі. Повторивши тест в циклі 100 разів, час етапу BUILD склав 211 секунд (рис. 5).

На наступному етапі було запущено виконання ста тестів в паралельному режимі засобами AWS CodeBuild, і результати виявились неочікуваними. Час виконання в стані BUILD збільшився до 765 секунд. Для вирішення ситуації з CodeBuild одним з варіантів розв'язку проблеми є застосування алгоритмів штучного інтелекту [14], але швидший спосіб використати оптимізацію на рівні операційної системи засобами Python.

Build logs | Phase details | Reports | Environment variables | Build details | Resource utilization

Name	Status	Context	Duration
SUBMITTED	Succeeded	-	<1 sec
QUEUED	Succeeded	-	<1 sec
PROVISIONING	Succeeded	-	4 secs
DOWNLOAD_SOURCE	Failed	CLIENT_ERROR: authorization failed for primary source	95 secs
FINALIZING	Succeeded	-	<1 sec
COMPLETED	Succeeded	-	-

Рис. 3. Помилка в стані DOWNLOAD_SOURCE

Name	Status	Context	Duration
SUBMITTED	Succeeded	-	<1 sec
QUEUED	Succeeded	-	<1 sec
PROVISIONING	Succeeded	-	5 secs
DOWNLOAD_SOURCE	Succeeded	-	4 secs
INSTALL	Succeeded	-	<1 sec
PRE_BUILD	Failed	COMMAND_EXECUTION_ERROR: Error while executing command: pip install -r requirements.txt. Reason: exit status 1	23 secs
FINALIZING	Succeeded	-	<1 sec
COMPLETED	Succeeded	-	-

Рис. 4. Помилка в стані PRE_BUILD

Name	Status	Context	Duration
SUBMITTED	Succeeded	-	<1 sec
QUEUED	Succeeded	-	<1 sec
PROVISIONING	Succeeded	-	5 secs
DOWNLOAD_SOURCE	Succeeded	-	139 secs
INSTALL	Succeeded	-	1 sec
PRE_BUILD	Succeeded	-	<1 sec
BUILD	Succeeded	-	211 secs
POST_BUILD	Succeeded	-	<1 sec
UPLOAD_ARTIFACTS	Succeeded	-	<1 sec
FINALIZING	Succeeded	-	<1 sec
COMPLETED	Succeeded	-	-

Рис. 5. Послідовне виконання тестів

Name	Status	Context	Duration
SUBMITTED	Succeeded	-	<1 sec
QUEUED	Succeeded	-	<1 sec
PROVISIONING	Succeeded	-	5 secs
DOWNLOAD_SOURCE	Succeeded	-	163 secs
INSTALL	Succeeded	-	3 secs
PRE_BUILD	Succeeded	-	<1 sec
BUILD	Succeeded	-	73 secs
POST_BUILD	Succeeded	-	<1 sec
UPLOAD_ARTIFACTS	Succeeded	-	<1 sec
FINALIZING	Succeeded	-	<1 sec
COMPLETED	Succeeded	-	-

Рис. 6. Паралельне виконання тестів засобами Python

Для реалізації було використано альтернативний спосіб запуску тестів. До процесу тестування додана можливість запуску в багатопроцесорному режимі і запущено той же тест 100 разів. Результат тестування виправдав очікування – час етапу Build зменшився до 73 секунд (рис. 6).

Висновки. У роботі змодельовано основні етапи процесу збірки, включаючи паралельне виконання тестів. Модель побудовано засобами скінченного часового автомату та мережі Петрі. Представлено модифікований алгоритм розгортання програмного забезпечення з використанням багатопоточності, що дозволяє оптимізувати процес збірки та тестування в середовищі AWS CodeBuild, та усунути обмеження стандартних механізмів паралелізації AWS CodeBuild.

Результати дослідження показали, що використання стандартних засобів паралельного тестування в AWS CodeBuild може призводити до збільшення часу виконання. Натомість впровадження модифікованого алгоритму з багатопоточністю дозволяє значно скоротити загальний час розгортання.

Розроблений алгоритм є особливо актуальним для великих програмних проектів, які вимагають частих ітераційних збірок та тестування. Отримані результати можуть бути використані для вдосконалення автоматизованих процесів розгортання та управління обчислювальними ресурсами у хмарних середовищах.

Список використаної літератури

1. Shakhmetova G., Saukhanova Z., Udzir N. I., Sharipbay A., Saukhanov N. Application of Pseudo-Memory Finite Automata for Information Encryption. *IntelITSIS*. 2021. P. 330–339.
2. Cotumaccio N., Prezza N. On indexing and compressing finite automata. *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Philadelphia: Society for Industrial and Applied Mathematics, 2021. P. 2585–2599.
3. Jovanović N., Miljković D., Stamenković S., Jovanović Z., Chakraborty P. Teaching concepts related to finite automata using ComVis. *Computer Applications in Engineering Education*. 2021. Vol. 29, no. 5. P. 994–1006.
4. Nikulina O. M., Khatsko K.O. Method of converting the monolithic architecture of a front-end application to microfrontends. *Вісник Національного технічного університету «ХПІ». Серія: Системний аналіз, управління та інформаційні технології*. Харків: НТУ «ХПІ», 2023. № 2 (10). С. 79–84. DOI: doi.org/10.20998/2079-0023.2023.02.12.
5. Stotts P. D., Pugh W. Parallel finite automata for modeling concurrent software systems. *Journal of Systems and Software*. 1994. Vol. 27, no. 1. P. 27–43.
6. Liao H., Wang Y., Cho H. K., Stanley J., Kelly T., Lafortune S., et al. Concurrency bugs in multithreaded software: Modeling and analysis using Petri nets. *Discrete Event Dynamic Systems*. 2013. Vol. 23. P. 157–195.
7. Sen K., Viswanathan M. Model checking multithreaded programs with asynchronous atomic methods. *Computer Aided Verification: 18th International Conference, Seattle, WA, USA, August 17–20, 2006*. Springer, 2006. P. 300–314.
8. Philippe J., Omond A., Coullon H., Prud'Homme C., Raïs I. Fast Choreography of Cross-DevOps Reconfiguration with Ballet: A Multi-Site OpenStack Case Study. *International Conference on Software Analysis, Evolution and Reengineering*. 2024. P. 1–11. DOI: doi.org/10.1109/SANER60148.2024.00007.
9. Bo C., Dang V., Xie T., Wadden J., Stan M., Skadron K. Automata processing in reconfigurable architectures: In-the-cloud deployment, cross-platform evaluation, and fast symbol-only reconfiguration. *ACM Transactions on Reconfigurable Technology and Systems*. 2019. Vol. 12, no. 2. P. 1–25.
10. Zamkovyi M., Gavrylenko S., Khatsko K., Khatsko N., Algorithmic support for building a distributed iot system in a cloud service. *4th KhPI Week on Advanced Technology*, Kharkiv, Ukraine, 2023. P. 1–6. DOI: doi.org/10.1109/KhPIWeek61412.2023.1031299.

11. Zimba A., Hongsong C. Analyzing trust concerns in public clouds using finite state automata. *2nd International Conference on Cloud Computing and Internet of Things*. 2016. P. 25–29.
12. Vayadande K. B., Sheth P., Shelke A., Patil V., Shevate S., Sawakare C. Simulation and testing of deterministic finite automata machine. *International Journal of Computer Sciences and Engineering.* 2022. Vol. 10, no. 1. P. 13–17.
13. Alur R., Dill D. L. A theory of timed automata. *Theoretical Computer Science*. 1994. Vol. 126, no. 2. P. 183–235.
14. Gavrylenko S., Zozulia V., Khatsko N., Methods for improving the quality of classification on imbalanced data. *2023 IEEE 4th KhPI Week on Advanced Technology*, Kharkiv, Ukraine, 2023. P. 1–5. DOI: doi.org/10.1109/KhPIWeek61412.2023.10312879.

References (transliterated)

1. Shakhmetova G., Saukhanova Z., Udzir N. I., Sharipbay A., Saukhanov N. Application of Pseudo-Memory Finite Automata for Information Encryption. *IntelITSIS*. 2021, pp. 330–339.
2. Cotumaccio N., Prezza N. On indexing and compressing finite automata. *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Philadelphia: Society for Industrial and Applied Mathematics, 2021, pp. 2585–2599.
3. Jovanović N., Miljković D., Stamenković S., Jovanović Z., Chakraborty P. Teaching concepts related to finite automata using ComVis. *Computer Applications in Engineering Education*. 2021, vol. 29, no. 5, pp. 994–1006.
4. Nikulina O. M., Khatsko K.O. Method of converting the monolithic architecture of a front-end application to microfrontends. *Visnyk Natsional'noho tekhnichnoho universytetu "KhPI". Seriya: Systemnyy analiz, upravlinnya ta informatsiyni tekhnolohiyi.* [Bulletin of the National Technical University "KhPI". Series: System analysis, control and information technology]. Kharkov, NTU "KhPI" Publ., 2023, no. 2 (10), pp. 79–84. DOI: doi.org/10.20998/2079-0023.2023.02.12.
5. Stotts P. D., Pugh W. Parallel finite automata for modeling concurrent software systems. *Journal of Systems and Software.* 1994, vol. 27, no. 1, pp. 27–43.
6. Liao H., Wang Y., Cho H. K., Stanley J., Kelly T., Lafortune S., et al. Concurrency bugs in multithreaded software: Modeling and analysis using Petri nets. *Discrete Event Dynamic Systems*. 2013, vol. 23, pp. 157–195.
7. Sen K., Viswanathan M. Model checking multithreaded programs with asynchronous atomic methods. *Computer Aided Verification: 18th International Conference, Seattle, WA, USA, August 17–20, 2006*. Springer, 2006, pp. 300–314.
8. Philippe J., Omond A., Coullon H., Prud'Homme C., Raïs I. Fast Choreography of Cross-DevOps Reconfiguration with Ballet: A Multi-Site OpenStack Case Study. *International Conference on Software Analysis, Evolution and Reengineering*. 2024, pp.1–11. DOI: doi.org/10.1109/SANER60148.2024.00007.
9. Bo C., Dang V., Xie T., Wadden J., Stan M., Skadron K. Automata processing in reconfigurable architectures: In-the-cloud deployment, cross-platform evaluation, and fast symbol-only reconfiguration. *ACM Transactions on Reconfigurable Technology and Systems*. 2019, vol. 12, no. 2, pp. 1–25.
10. Zamkovyi M., Gavrylenko S., Khatsko K., Khatsko N., Algorithmic support for building a distributed iot system in a cloud service. *4th KhPI Week on Advanced Technology*, Kharkiv, Ukraine, 2023, pp. 1–6. DOI: doi.org/10.1109/KhPIWeek61412.2023.1031299
11. Zimba A., Hongsong C. Analyzing trust concerns in public clouds using finite state automata. *2nd International Conference on Cloud Computing and Internet of Things*. 2016, October, pp. 25–29.
12. Vayadande K. B., Sheth P., Shelke A., Patil V., Shevate S., Sawakare C. Simulation and testing of deterministic finite automata machine. *International Journal of Computer Sciences and Engineering.* 2022, vol. 10, no. 1, pp. 13–17.
13. Alur R., Dill D. L. A theory of timed automata. *Theoretical Computer Science*. 1994, vol. 126, no. 2, pp. 183–235.
14. Gavrylenko S., Zozulia V., Khatsko N., Methods for improving the quality of classification on imbalanced data. *2023 IEEE 4th KhPI Week on Advanced Technology*, Kharkiv, Ukraine, 2023, pp. 1–5. DOI: doi.org/10.1109/KhPIWeek61412.2023.10312879.

Надійшла (received) 05.12.2024

UDC 004.9

N. Y. KHATSKO, Candidate of Technical Sciences, Associate Professor, Associate Professor of the Department of Information Systems and Technologies of the National Technical University "Kharkiv Polytechnic Institute", Kyiv, Ukraine; e-mail: nataliia.khatsko@khpi.edu.ua; ORCID: https://orcid.org/0000-0002-2543-0280
M. V. SLIEPUSHKOV, Master's Student, National Technical University "Kharkiv Polytechnic Institute", Kharkiv, Ukraine; e-mail: m.sliepushkov@gmail.com; ORCID: https://orcid.org/0009-0001-0004-2820
K. O. KHATSKO, Senior Lecturer of the Department of the Department of Information Systems and Technologies of the National Technical University "Kharkiv Polytechnic Institute", Graduate Student, Kharkiv, Ukraine; e-mail: kyrylo.khatsko@khpi.edu.ua; ORCID: https://orcid.org/0000-0003-3315-1553
Y. O. SHEBANOV, Graduate Student, National Technical University "Kharkiv Polytechnic Institute", Kharkiv, Ukraine; e-mail: yevhenii.shebanov@cs.khpi.edu.ua; ORCID: https://orcid.org/0009-0006-9032-8764

MODIFIED SOFTWARE DEPLOYMENT ALGORITHM USING MULTI-THREADING

The article presents a modified deployment algorithm for software systems using multithreading in AWS CodeBuild, aimed at optimizing build time and reducing computational resource costs in cloud environments. The key stages of the build process, including parallel test execution, task allocation analysis, and resource management, were modeled using finite automata, timed automata, and Petri nets. Particular attention was given to identifying and addressing the limitations of AWS CodeBuild's standard parallelization mechanisms, which can lead to inefficient resource utilization and extended build durations. The study revealed that AWS CodeBuild's default mechanisms are not always capable of optimally leveraging system resources, especially when handling large software projects with numerous dependencies. To overcome these limitations, the use of Python's multithreading capabilities was proposed as a convenient tool for extending the platform's base functionality. The proposed approach enabled flexible thread management and task distribution at the user scenario level, significantly reducing overall build time. Experimental results demonstrated substantial reductions in build execution time compared to the default AWS CodeBuild settings, confirming the effectiveness of the proposed algorithm in enhancing performance and ensuring high scalability for build processes in cloud environments. The developed algorithm is particularly relevant for large software projects requiring frequent iterative builds and testing. The findings can be utilized to improve automated deployment processes and computational resource management in cloud ecosystems.

Keywords: deployment process, information technology, algorithm, software deployment process model, timed finite automaton, Petri nets.

Повні імена авторів / Author's full names

Автор 1 / Author 1: Хацько Наталія Євгенівна / Khatsko Nataliia Yevgenivna
Автор 2 / Author 2: Слєпушков Микола Васильович / Sliepushkov Mykola Vasylovych
Автор 3 / Author 3: Хацько Кирило Олександрович, Khatsko Kyrylo Olexandrovych
Автор 4 / Author 4: Шебанов Євгеній Олександрович / Shebanov Yevhenii Olexandrovych