УДК 004.415.2
DOI https://doi.org/10.32782/2663-5941/2024.1.1/43

Пфайфер В.В.
Національний університет «Львівська політехніка»

Бешлей М.І.
Національний університет «Львівська політехніка»

Селюченко М.О.
Національний університет «Львівська політехніка»

Брич М.В.
Національний університет «Львівська політехніка»

Климаш М.М.
Національний університет «Львівська політехніка»

АВТОМАТИЗАЦІЯ ОНОВЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ РОЗПОДІЛЕНИХ ІНФОКОМУНІКАЦІЙНИХ СИСТЕМ

У межах даної роботи увагу сфокусовано навколо питання оновлення ПЗ, зокрема таких компонентів, як бібліотеки, модулі, та фреймворки, оскільки їх використання при написанні коду стало загальною практикою в сучасній розробці завдяки таким перевагам як повторне використання коду, модульність, стандартизація коду, та підвищення швидкості розробки. Проте, попри переваги, використання залежностей має й свої ризики – конфлікти версій, проблеми сумісності при оновленні, залежність від сторонніх сервісів, прогалини в безпеці.

У роботі проведено аналітичний огляд ключових аспектів життєвого циклу розробки програмного забезпечення (SDLC) та методології безперервної інтеграції та розгортання змін (CI/CD), проаналізовано ключові аспекти тестування продуктивності програмних інтерфейсів (API), та досліджено концепцію залежностей від коду (Code Dependencies) і процес їх оновлення загальноприйнятим методом. Удосконалено метод оновлення компонентів ПЗ, шляхом повної автоматизації кроків, та розроблено відповідний сценарій для автоматизації процесу в середовищі GitHub Actions, і досліджено виконання удосконаленого методу. Проведено порівняльний аналіз ефективності ручного, напівавтоматизованого та повністю автоматизованого методів. Експериментальні результати підтверджують, що розроблений повністю автоматизований метод оновлення залежностей ПЗ в контексті розробленого тестового середовища є на 36% ефективнішим від напівавтоматичного методу та виключає активну участь розробника, що зменшує його витрати часу на більше ніж 90%. Повна автоматизація в свою чергу виключає помилку внаслідок людського фактору, що гаратнує стабільну продуктивність та безпеку функціонування розподілених інформаційно комунікаційних систем.

Ключові слова: *CI/CD, Залежності, Автоматизація розгортання ПЗ, Оновлення ПЗ.*

Постановка проблеми. Оновлення програмного забезпечення це невід'ємна і надзвичайно важлива складова його життєвого циклу, оскільки забезпечує виправлення помилок, вдосконалення безпеки та впровадження нових функцій. Особливо це важливо для програмного забезпечення, яке відповідає за функціонування та управління складними розподіленими системами які повсюдно використовуються, зокрема, програмні маршрутизатори чи комутатори, контролери мереж, шлюзи, фаєрволи, сервери, елементи ядра мобільних систем, такі як реєстри, пакетні комутатори, білінг системи і т.д. Процес розробки і оновлення ПЗ неможливий без його розгортання, який базується на методах безперервної інтеграції та безперервного розгортання, або CI/CD (Continuous Integration & Continuous Delivery). Звичною практикою в сучасній розробці програмного забезпечення стало використання залежностей – сторонніх бібліотек, які є по суті окремим ПЗ, яке також повинно постійно розвиватися, щоб відповідати стандартам безпеки і залишатися актуальним. Одним з недоліків поширеного в IT галузі методу CI/CD є неможливість визначити які саме оновлення спричинили раптове зниження продуктивності розподіленої

системи. Для критичних систем управління, така ситуація неприпустима, адже одразу призведе до збільшення часу прийняття рішень та погіршення продуктивності, доступності та якості обслуговування користувачів. У випадках коли критичне ПЗ потребує термінового оновлення через те, що необхідно виправити критичну несправність чи виявлену прогалину в безпеці, тривалість розробки, тестування та розгортання повинні бути мінімізовані, для того, щоб гарантувати стабільну, надійну та безпечну роботу всієї розподіленої системи. В цьому полягає ще один суттєвий недолік існуючого методу CI/CD, оскільки у випадку погіршення продуктивності ПЗ після оновлення, тривалість пошуку та виправлення причини, яка призвела до уповільнення роботи розподіленої системи, залежить від ручного процесу який покладається на розробників системи.

Аналіз останніх досліджень і публікацій. У великих та складних проєктах правильне керування залежностями є важливим завданням, що вимагає системного підходу до планування, вибору та підтримки залежностей для забезпечення надійності та ефективності програмного забезпечення [1]. Використання інструменту керування залежностями заощаджує час і ресурси, а також робить код більш портативним і відтворюваним [3]. Деякі приклади інструментів керування залежностями: npm для JavaScript, pip для Python, Maven для Java та Bundler для Ruby. Використання зовнішніх залежностей в програмному забезпеченні несе ризики, особливо у випадку оновлення цих залежностей у застарілих проєктах. У такому випадку може виникнути ефект доміно через проблеми сумісності, коли оновлення однієї бібліотеки може вимагати оновлення інших, через помилки або проблеми з безпекою. У разі зміни API цих залежностей, це може вимагати повного переписування програмного коду. У великих корпоративних програмах часто залишаються невикористані залежності у файлі маніфесту, навіть якщо вони більше не використовуються у коді. Це може призвести до збільшення розміру двійкових файлів, займати більше ресурсів та часу при запуску програми. Також можливі конфлікти між бібліотеками при додаванні нових залежностей, а також використання застарілих бібліотек, які містять помилки або проблеми з безпекою. Проблеми сумісності при оновленні бібліотек та вплив на продуктивність через збільшений обсяг коду можуть стати суттєвими [2]. Попередні дослідження проведені на аналізі 2700 бібліотек для Java у 4600 проєктах на платформі GitHub показало, що 81,5% залишаються застарілими, і містять проблеми пов'язані з безпекою [4]. Інше з досліджень виявило, що такі фактори, як невизначеність при оцінці зусиль для рефакторингу, інші пріоритети завдань, та відповідальність за ризики, служать причинами, з яких розробники не оновлюють залежності [5].

Метою роботи є забезпечити надійну, стабільну та безпечну роботу розподілених інформаційно-комунікаційних систем шляхом автоматизації процесу оновлень компонентів програмного забезпечення.

Виклад основного матеріалу дослідження. Перш за все необхідно дослідити, як здійснюється метод оновлення залежностей програмного забезпечення вручну та яка його ефективність. Для цього здійснено наступні кроки:

- Розроблено веб застосунок для оплати послуг онлайн, побудований на базі фреймворку Express для Node.js.
- Налаштовано Azure WebApps як платформу для розгортання та виконання розробленого веб застосунку.
- За допомогою Cypress розроблено комплекс автоматизованих тестів, що охоплює три рівні тестування: модульні тести (unit tests / component tests); тести користувацького інтерфейсу (e2e tests); тести продуктивності (performance tests).
- Для проведення тестів продуктивності додатку налаштовано інструмент тестування якості веб-сайтів Lighthouse.
- Створено репозиторій з кодом веб застосунку та всіма розробленими автоматизованими тестами на платформі GitHub.

В процесі аналізу та декомпозиції виокремлено 9 основних етапів методу оновлення залежностей програмного забезпечення, а також поставлено у відповідність кроки, які необхідно здійснити в контексті розробленого нами веб застосунку для оновлення його залежностей з використанням налаштованих інструментів (табл. 1).

В ході проведеного експерименту з використанням ручного методу всі кроки здійснено вручну, а саме аналіз поточного стану бібліотек ПЗ та їх вразливостей. Далі здійснено оновлення компонента ejs, що містить критичну вразливість. Після цього проведено розгортання коду у віддаленому середовищі, та ряд тестів, результати яких є вдалими. На останньому кроці було синхронізовано локальні зміни із віддаленим середовищем та злито зміни в гілку розробки. Увесь процес оновлення ручним методом триває 47 хвилин і потребує стільки ж часу розробника (табл. 2).

Таблиця 1

Послідовність кроків виконання експерименту

Крок	Команда
Перевірка актуального стану бібліотек	npm outdated
Аналіз наявних вразливостей	npm audit
Оновлення бібліотеки до нової версії	npm install
Розгортання коду у віддаленому середовищі	-
Виконання автоматизованих модульних тестів	cypress run --component
Виконання автоматизованих тестів користувацького інтерфейсу	cypress run --e2e –spec ’unitTests.cy.js’
Виконання автоматизованих тестів швидкодії	cypress run --e2e –spec ’perfTests.cy.js’
Аналіз результатів тестування	-
Злиття змін з гілкою розробки	git add git push

Таблиця 2

Затрати часу на виконання кроків поточним методом

Крок	Час, хв	
	Розробників	Загальний
Перевірка актуального стану бібліотек	5	5
Аналіз наявних вразливостей	5	5
Оновлення бібліотеки до нової версії	1	1
Розгортання коду у віддаленому середовищі	5	5
Виконання автоматизованих модульних тестів	10	10
Виконання автоматизованих тестів користувацького інтерфейсу	10	10
Виконання автоматизованих тестів швидкодії	5	5
Аналіз результатів тестування	5	5
Злиття змін з гілкою розробки	1	1

На рисунку 1 зображено діаграму, яка висвітлює послідовність процесів напівавтоматизованого методу, який виконується за методологією CI/CD.

У напівавтоматизованій версії методу ручного оновлення чотири кроки виконуються автоматично (етап тестування), а два перших – напівавтоматично (етап аналізу), інші три кроки: оновлення бібліотеки, аналіз результатів, злиття – виконуються вручну. Загальний час на оновлення залишився незмінним 47 хв., проте затрати часу розробника зменшилися до 19 хв. (табл. 3) завдяки частковій автоматизації процесів з використанням інструментів CI/CD. З практичної точки зору, затрати часу працівника на ініціалізацію й виконання автоматизованих кроків є рівними нулю, проте з ціллю коректної візуалізації отриманих результатів на графіках, приймемо, що кожен автоматизований крок займатие 0.5 хв. часу розробника. Цей час відведено для періодичного моніторингу успішності ходу виконання процесів.

У даній роботі удосконалено метод оновлення залежностей шляхом повної автоматизації всіх кроків ручного методу. Зокрема, використано ширший спектр функцій CI\CD, що дозволяють автоматизувати ручні дії: оновлення версії бібліотеки, аналіз результатів тестування, злиття гілок. Діаграму запропонованого рішення зображено на рисунку 2.

Таблиця 3

Затрати часу на виконання кроків напівавтоматизованим методом

Крок	Час розробників, хв	Час виконання, хв
Перевірка актуального стану бібліотек	5	5
Аналіз наявних вразливостей	5	5
Оновлення бібліотеки до нової версії	1	1
Розгортання коду у віддаленому середовищі	0.5	5
Виконання автоматизованих модульних тестів	0.5	10
Виконання автоматизованих тестів користувацького інтерфейсу	0.5	10
Виконання автоматизованих тестів швидкодії	0.5	5
Аналіз результатів тестування	5	5
Злиття змін з гілкою розробки	1	1

У порівнянні з розглянутими ручним та напівавтоматизованими методом (рис. 1), розроблений варіант дозволяє повністю автоматизувати кроки перевірки актуальності залежностей, вразливостей, та оновлення їх версій завдяки використанню відповідних сторонніх сервісів. У ролі такого сервісу використовується Dependabot, який містить набір необхідних функцій, а також є вбудованим у платформу GitHub.

Наступним покращенням є автоматизація аналізу результатів тестування, та ініціалізації злиття гілок. Це здійснюється шляхом модифікації наявного сценарію конвеєру, у який добавляється додаткова дія з використання функцій загальнодоступного розширення github-action-merge-dependabot.

Розглянемо хід виконання автоматизованих процесів реалізованого конвеєру. Спершу бот здій-

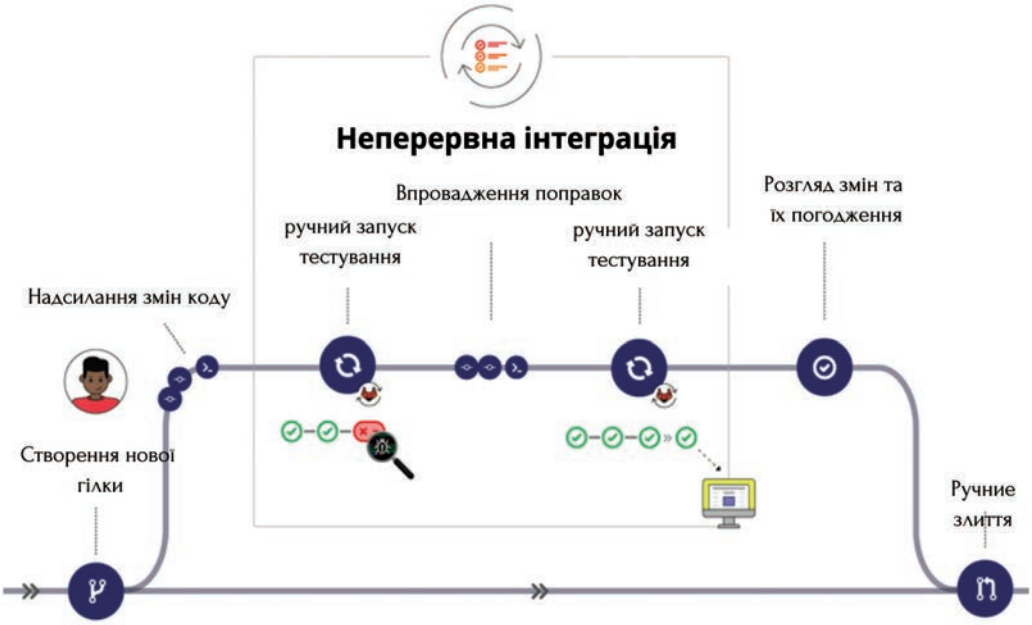

Рис. 1. Діаграма процесів напівавтоматизованого методу оновлення залежностей

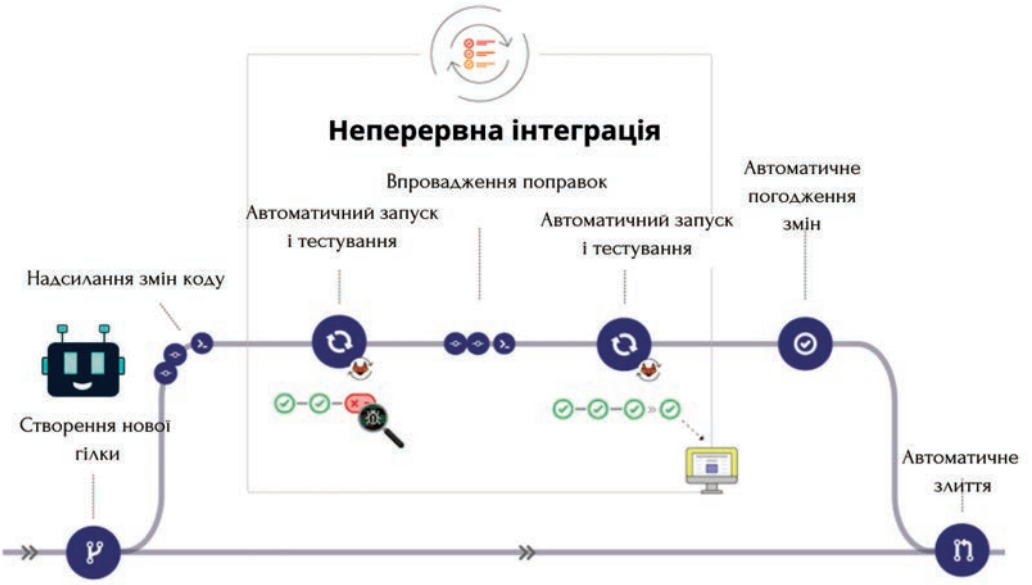

Рис. 2. Діаграма розробленого методу оновлення залежностей

снює аналіз версій бібліотек додатку, і відкриває пул-запити на оновлення тих, версії яких є застарілими (рис. 3). Конвеєр знаходиться в сплячому режимі, допоки Dependabot не ініціалізує новий пул-запит. Після відкриття пул-запитів розпочинається почергове виконання конвеєрів на відповідних гілках розробки (кожна з яких містить оновлення окремої бібліотеки), оскільки спрацьовує подія (on: pull-request).

Далі здійснюється виконання всіх вищезгаданих кроків. З рисунку 4 слідує, що розпочалося оновлення бібліотеки ejs (версія 2.6.2) до актуальної версії 3.1.9. Результати проходження модульних тестів є вдалими, генерується архів з оновленим кодом. Після чого оновлена версія застосунку розгортається у сервісі Azure WebAppp (доступ за посиланням https://payment-form-test-0101.azurewebsites.net), і починається наступний етап – тестування користувацького інтерфейсу та продуктивності додатку. Оскільки результати тестів є також вдалими, ініціалізується автоматичне об'єднання змін з гілкою development.

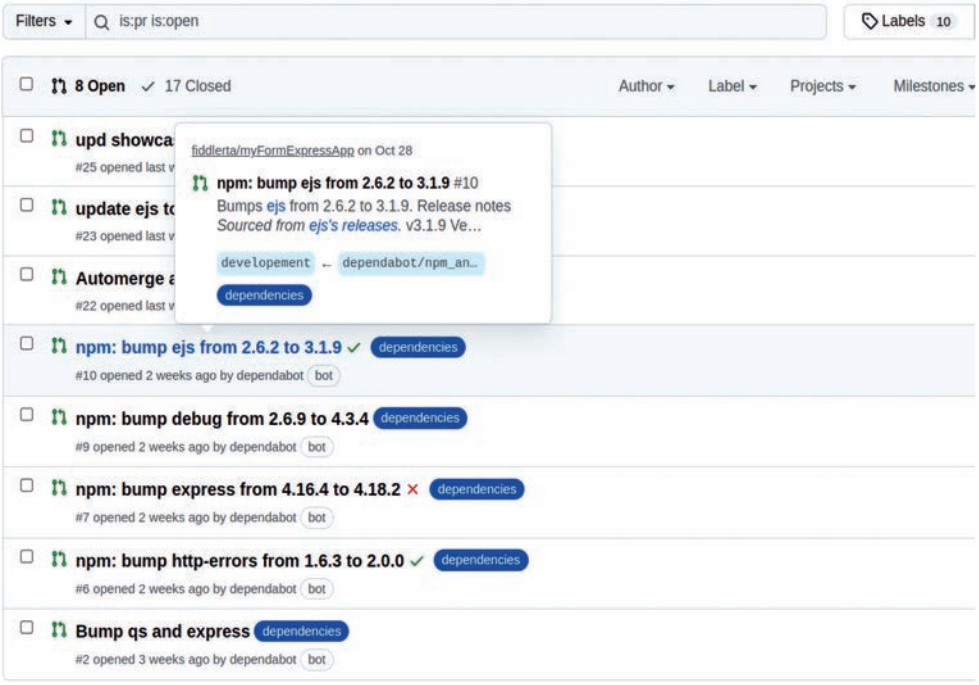

Рис. 3. Перелік пул-запитів відкритих ботом Dependabot

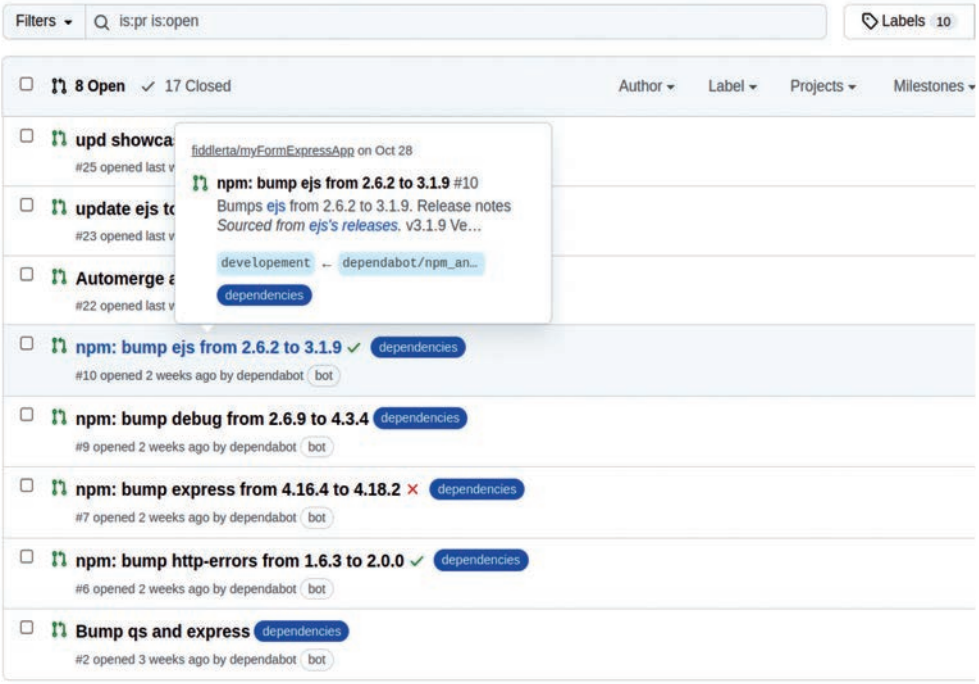

Рис. 4. Хід виконання процесів конвеєру для оновлення бібліотеки ejs

У ході успішного завершення всіх процесів, відбулося автоматичне об'єднання змін з гілки ejs із головною – development. Щоб переконатися в цьому відкриємо пул-запит гілки ejs (рис. 5) – як бачимо наразі запит об'єднано і закрито.

З результатів слідує, що побудований конвеєр на основі інтеграції з ботом Dependabot, сервісом Azure WebApp, та використанням можливостей GitHub Actions на платформі GitHub виконує поставлену ціль автоматичного аналізу та оновлення застарілих версій бібліотек програмного забезпечення.

В ході проведеного експерименту з використанням удосконаленого автоматизованого методу перші три етапи експерименту, а саме перевірка актуального стану бібліотек, аналіз існуючих

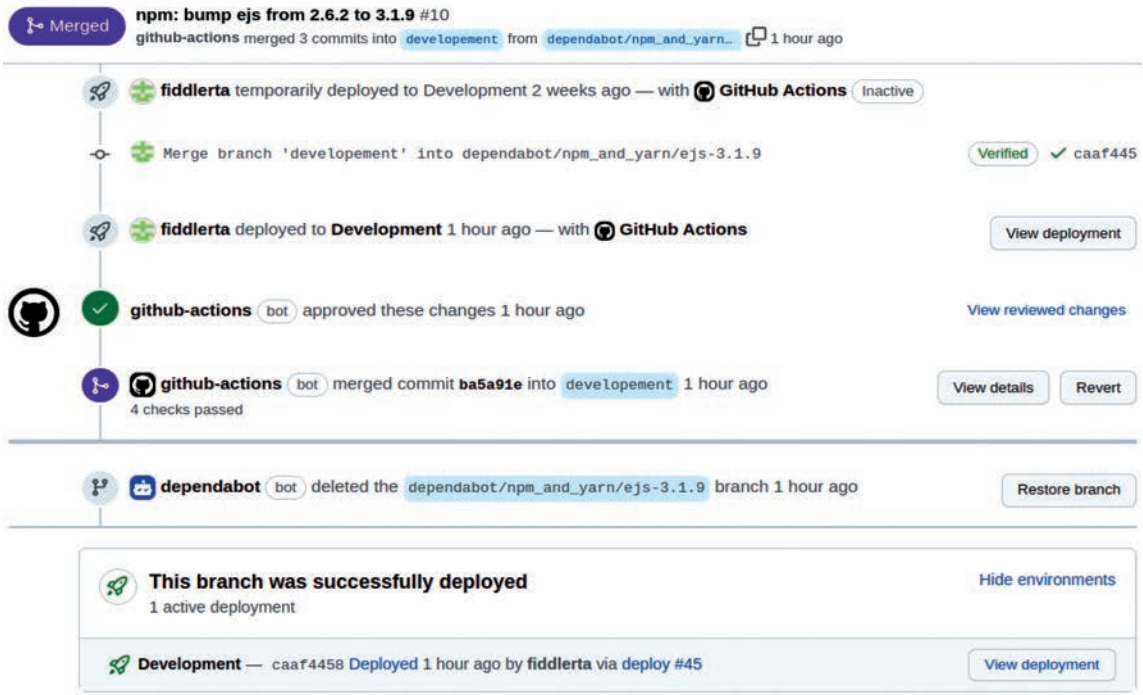

Рис. 5. Вікно пул-запиту для гілки з оновленням бібліотеки ejs

вразливостей та оновлення бібліотеки, виконуються автоматично за допомогою вбудованого сервісу Dependabot. Це дозволяє значно зекономити час розробника та забезпечити швидке реагування на зміни в екосистемі використовуваних бібліотек.

Кроки 4, 5, 6 та 7, які включають розгортання коду та виконання автоматизованих тестів, також виконуються автоматично, надаючи додаткову автоматизацію та покращення швидкості виконання цих етапів (таблиця 4).

Таблиця 4

Затрати часу на виконання кроків автоматизованим методом

Крок	Час розробників, хв	Час виконання, хв
Перевірка актуального стану бібліотек	0.5	0
Аналіз наявних вразливостей	0.5	0
Оновлення бібліотеки до нової версії	0.5	0
Розгортання коду у віддаленому середовищі	0.5	3
Виконання автоматизованих модульних тестів	0.5	10
Виконання автоматизованих тестів користувацького інтерфейсу	0.5	10
Виконання автоматизованих тестів швидкодії	0.5	5
Аналіз результатів тестування	0.5	0
Злиття змін з гілкою розробки	0.5	0.1

Останні два кроки, 8 та 9, що передбачають аналіз результатів тестування та злиття змін з гілкою розробки, тепер виконуються завдяки функціональності CI/CD платформи GitHub Actions. Ця дія автоматично перевіряє успішність пройдених тестів та, у випадку їх успішного виконання, ініціює автоматичне об'єднання гілки, створеної Dependabot, з головною гілкою розробки (development). Такий механізм значно збільшує ефективність та автоматизує фінальні кроки процесу розробки та тестування.

У цьому експерименті виконання дев'яти послідовних кроків для оновлення однієї залежності тривало близько 30 хв, що на 17 хв. хвилин менше ніж у двох попередніх експериментах, завдяки тому, що всі інші етапи крім тестування займають в середньому 3 хв., а середня тривалість проходження тестів 25–27 хв., проте повна автоматизація процесу зводить до нуля затрати часу розробника.

Порівняння ефективності розглянутих методів.

Порівнюючи результати проведених експериментів, можна визначити вагомі покращення у виконанні процесу оновлення залежностей у програмному забезпеченні при використанні автоматизації (табл. 5).

У першому експерименті, коли весь процес оновлення виконувався вручну, затрати часу розробника становили приблизно 47 хвилин.

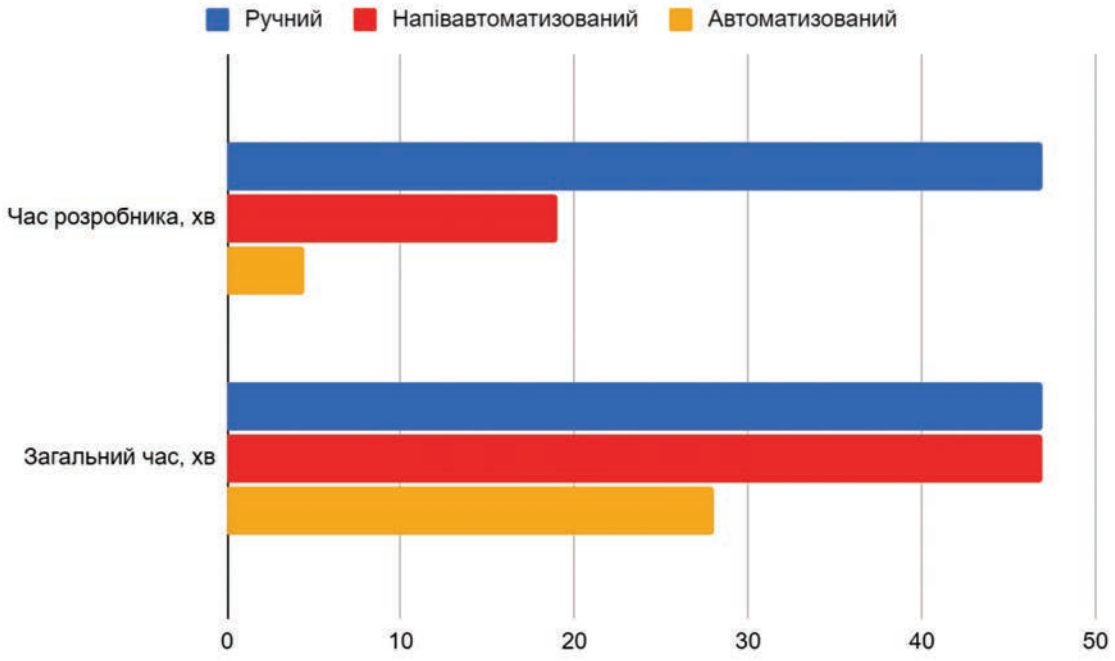

Рис. 6. Порівняльна діаграма витрат часу розробника в залежності від обраного методу

Таблиця 5

Порівняльна таблиця витрат часу розробників на виконання кроків розглянутими методами

Крок / Метод	Час розробника, хв			Загальний час, хв		
	Р	Н	А	Р	Н	А
Перевірка актуального стану бібліотек	5	5	0,5	5	5	0
Аналіз наявних вразливостей	5	5	0,5	5	5	0
Оновлення бібліотеки до нової версії	1	1	0,5	1	1	0
Розгортання коду у віддаленому середовищі	5	0,5	0,5	5	5	3
Виконання автоматизованих модульних тестів	10	0,5	0,5	10	10	10
Виконання автоматизованих тестів користувацького інтерфейсу	10	0,5	0,5	10	10	10
Виконання автоматизованих тестів швидкодії	5	0,5	0,5	5	5	5
Аналіз результатів тестування	5	5	0,5	5	5	0
Злиття змін з гілкою розробки	1	1	0,5	1	1	0.1
Сума	47	19	4.5	47	47	28

У другому експерименті, з використанням елементів автоматизації, час виконання всього процесу залишився на однаковому рівні, склавши також 47 хвилин. Однак важливо відзначити, що використання автоматизації знизило затрати часу розробника до 19 хвилин, що становить трохи більше ніж дві третини загального часу виконання процесу.

У третьому експерименті, де було впроваджено повністю автоматизований метод, виділяється радикальне покращення. Затрати часу розробника стали нульовими, оскільки усі кроки автоматизовано. Важливо врахувати, що для візуалізації результатів на графіку прийнято, що кожен крок займає 0.5 хвилини, а отже, сумарні витрати складають лише 4.5 хвилин.

На рис. 6 показано, що впровадження повністю автоматизованого методу призвело до значного зменшення часових затрат розробника, підвищивши ефективність та швидкість виконання процесу оновлення залежностей у програмному забезпеченні.

Висновки. Розроблений автоматизований метод оновлення програмного забезпечення на 36% зменшує загальну тривалість оновлення програмних компонентів та на 90% зменшує залучення розробника в процес оновлення. В контексті інфокомунікаційних систем та мереж розроблений метод дозволяє оперативно реагувати на випуск нових версій бібліотек та швидко впроваджувати їх у систему. Це сприяє підтримці безпеки мережевого обладнання та забезпеченню відповідності

останнім стандартам безпеки. Автоматичне впровадження змін також допомагає зменшити кількість рутинної роботи та ризик помилок, що можуть виникнути при ручному оновленні. За рахунок автоматизованого процесу досягається підвищення ефективності та зниження часових витрат. Ці фактори стають важливими в контексті інфо-комунікаційних систем, де оперативність та надійність є ключовими вимогами для забезпечення безперебійного функціонування системи.

Список літератури:

1. Sonatype: What Are Software Dependencies? // електрон. текст. дані URL: https://www.sonatype.com/launchpad/what-are-software-dependencies

2. Snyk.io: Best practices for managing Java dependencies // електрон. текст. дані URL: https://snyk.io/blog/best-practices-for-managing-java-dependencies/

3. Linkedin.com: How do you handle complex and interdependent software components and dependencies? // електрон. текст. дані URL: https://www.linkedin.com/advice/0/how-do-you-handle-complex-interdependent-software-components

4. Do developers update their library dependencies? / R. G. Kula et al. Empirical Software Engineering. 2017. Vol. 23, no. 1. P. 384–417. URL: https://doi.org/10.1007/s10664-017-9521-5

5. Hejderup J., Gousios G. Can we trust tests to automate dependency updates? A case study of java projects. Journal of Systems and Software. 2021. P. 111097. URL: https://doi.org/10.1016/j.jss.2021.111097.

Pfaifer V.M., Beshley M.I., Seliuchenko M.O., Brych M.V., Klymash M.M. **AUTOMATION OF THE SOFTWARE UPDATE OF DISTRIBUTED INFOCOMMUNICATION SYSTEMS**

Within the framework of this work, attention is focused around the issue of updating software, in particular such components as libraries, modules, and frameworks, since their use when writing code has become a common practice in modern development due to such advantages as code reuse, modularity, code standardization, and improvement speed of development. However, despite the advantages, the use of dependencies has its own risks – version conflicts, compatibility problems during updating, dependence on third-party services, security gaps.

The paper provides an analytical review of key aspects of the software development life cycle (SDLC) and continuous integration and change deployment (CI/CD) methodology, analyzes key aspects of API performance testing, and explores the concept of code dependencies (Code Dependencies) and the process of updating them by a generally accepted method.

The method of updating software components has been improved by fully automating the steps, and a corresponding script has been developed to automate the process in the GitHub Actions environment, and the implementation of the improved method has been investigated. A comparative analysis of the effectiveness of manual, semi-automated and fully automated methods was carried out. Experimental results confirm that the developed fully automated method of updating software dependencies in the context of the developed test environment is 36% more efficient than the semi-automatic method and excludes the active participation of the developer, which reduces his time consumption by more than 90%. Full automation, in turn, excludes errors due to the human factor, which guarantees stable performance and safety of distributed information and communication systems.

Key words: *CI/CD, Dependencies, Software Deployment Automation, Software Update.*