УДК 004.42

Н. І. ТРУФАНОВ, А. І. ПОВОРОЗНЮК
Національний технічний університет «Харківський політехнічний інститут»

ПЕРСПЕКТИВИ ЗАСТОСУВАННЯ РЕАКТИВНОГО ПРОГРАМУВАННЯ ПРИ РОЗРОБЦІ ВЕБСЕРВІСІВ

У статті здійснено комплексний аналіз реактивного програмування як сучасної парадигми, що набуває все більшої популярності у сфері розробки вебсервісів. Реактивний підхід базується на принципах асинхронної обробки даних, подієво-орієнтованої архітектури та неблокуючих операцій, що дозволяє ефективно вирішувати проблеми масштабованості, стійкості та продуктивності у розподілених системах. Особливу увагу приділено питанням забезпечення швидкого відгуку системи в умовах високих навантажень та частих збоїв, що є актуальним для бізнесу, який прагне відповідати сучасним вимогам до швидкості та якості обслуговування користувачів. У статті проведено порівняльний аналіз реактивного та імперативного підходів до розробки вебсервісів, визначено основні переваги реактивної парадигми, серед яких підвищення ефективності використання ресурсів, покращення користувацького досвіду, зменшення часу відгуку та підвищення стійкості до збоїв. Окремо розглянуто основні виклики впровадження реактивного програмування, зокрема складність освоєння нових концепцій, збільшення складності архітектури, труднощі у знаходженні помилок асинхронних процесів, підвищені вимоги до моніторингу та діагностики, а також необхідність ретельного управління ресурсами. Висвітлено сучасні тенденції розвитку реактивних фреймворків і бібліотек, таких як Project Reactor, RxJava, Akka Streams, Spring WebFlux та RSocket, які забезпечують розробникам інструменти для побудови масштабованих, гнучких і стійких до збоїв систем. Проаналізовано практичні аспекти впровадження реактивного підходу, зокрема вплив на продуктивність, стабільність, операційні витрати та підтримку системи. Визначено сфери, де реактивне програмування є найбільш ефективним, а також окреслено ситуації, у яких традиційні імперативні підходи залишаються доцільними. Окремо підкреслено, що підвищена продуктивність, яку забезпечує реактивне програмування, не завжди є критичною для більшості сервісів, і у багатьох випадках достатньо використовувати прості та перевірені архітектурні підходи, які легше впроваджувати та підтримувати. Реактивні системи демонструють себе найкраще в середовищах з високим навантаженням, де необхідна масштабованість і стійкість до збоїв. Якщо система працює у стабільному режимі з невеликою кількістю одночасних запитів, переваги реактивної парадигми можуть бути незначними або навіть нівелюватися. Таким чином, реактивне програмування не є універсальним рішенням, а потужним інструментом для вирішення специфічних завдань сучасної цифрової інженерії. Вибір цієї парадигми має базуватися на реальних потребах проєкту, характері навантаження, вимогах до масштабування та стійкості, а також готовності команди до освоєння нових підходів. Принципи реактивного програмування, ймовірно, стануть ще більш актуальними у майбутньому, забезпечуючи фундамент для створення наступного покоління вебсервісів, які відповідатимуть вимогам цифрової епохи щодо продуктивності, стійкості та масштабованості.

Ключові слова: *реактивне програмування, вебсервіси, асинхронна обробка, подієво-орієнтована архітектура, backpressure, масштабованість, стійкість, Project Reactor, RxJava, Akka Streams, Spring WebFlux, RSocket.*

N. I. TRUFANOV, A. I. POVOROZNYUK
National Technical University “Kharkiv Polytechnic Institute”

PROSPECTS OF USING REACTIVE PROGRAMMING IN WEB SERVICES DEVELOPMENT

The article presents a comprehensive analysis of reactive programming as a modern paradigm that is gaining increasing popularity in the field of web service development. The reactive approach is based on the principles of asynchronous data processing, event-driven architecture, and non-blocking operations, which enable effective solutions to the challenges of scalability, resilience, and performance in distributed systems. Special attention is paid to ensuring rapid system responsiveness under high loads and frequent failures, which is crucial for businesses seeking to meet contemporary requirements for speed and quality of user service. The article provides a comparative analysis of reactive and imperative approaches to web service development, identifying the main advantages of the reactive paradigm, including improved resource utilization, enhanced user experience, reduced response times, and increased fault tolerance. The main challenges of adopting reactive programming are also discussed, such as the steep learning curve, increased architectural complexity, difficulties in debugging asynchronous processes, higher requirements for monitoring and diagnostics, and the need for careful resource management. The article highlights current trends in the development of reactive frameworks and libraries, such as Project Reactor, RxJava, Akka Streams, Spring WebFlux, and RSocket, which provide

https://doi.org/10.32782/mathematical-modelling/2026-9-1-36 ISSN 2618-0332 (print), 2618-0340 (online)

developers with tools for building scalable, flexible, and resilient systems. Practical aspects of implementing the reactive approach are analyzed, including its impact on performance, stability, operational costs, and system maintenance. The areas where reactive programming is most effective are determined, and situations in which traditional imperative approaches remain more appropriate are outlined. It is emphasized that the increased performance provided by reactive programming is not always critical for most services, and in many cases, it is sufficient to use simple and proven architectural approaches that are easier to implement and maintain. Reactive systems perform best in high-load environments where scalability and fault tolerance are required. If the system operates in a stable mode with a small number of concurrent requests, the advantages of the reactive paradigm may be insignificant or even negated. Thus, reactive programming is not a universal solution but a powerful tool for solving specific problems of modern digital engineering. The choice of this paradigm should be based on the actual needs of the project, the nature of the workload, scalability and resilience requirements, and the team's readiness to master new approaches. The principles of reactive programming are likely to become even more relevant in the future, providing a foundation for the next generation of web services that meet the demands of the digital era for performance, resilience, and scalability.

Keywords: *reactive programming, web services, asynchronous processing, event-driven architecture, backpressure, scalability, resilience, Project Reactor, RxJava, Akka Streams, Spring WebFlux, RSocket.*

Постановка проблеми

У теперішніх умовах цифровізації швидкість обробки та отримання результату є критичним фактором, який може виділяти продукт від своїх конкурентів. Вебсервіси, що мають велику кількість взаємодій з користувачами, повинні відповідати високим вимогам продуктивності та відгуку системи. І це також стосується вебсервісів, до прикладу подумайте чи готові ви чекати більше 30 секунд для отримання результату після натискання кнопки на сайті, скоріш за все відповідь ні.

Через це і постає питання: чи здатна парадигма реактивного програмування забезпечити значні переваги у порівнянні з традиційними імперативними підходами до розробки вебсервісів? У цій статті розглядаються можливості реактивного програмування для підвищення ефективності, масштабованості та стійкості сучасних вебсистем, порівняння його з імперативними системами.

Аналіз останніх досліджень і публікацій

Проаналізувавши останні публікації присвячені реактивному програмуванню приходимо до висновку, що єдина думка щодо доцільності та перспективи цієї парадигми відсутня. Частина авторів стверджує, що реактивне програмування поступово втрачає актуальність через складність впровадження, зростання альтернативних технологій або недостатню кількість переваг у порівнянні з стандартними підходами. Як приклад можна взяти висновок автора, за якого, поява віртуальних потоків у Java 21 та розвиток сучасних фреймворків знижують потребу у реактивних рішеннях [1; 2], а деякі розробники навіть відмовляються від використання реактивної архітектури через складність підтримки та відсутність значних переваг у їхніх проєктах [3].

У цей же час існує велика кількість статей в підтримку реактивного програмування, де автори вважають це архітектурне рішення майбутнім для вебсервісів [4] або якісним підходом, який у правильній ситуації сильно обганяє інші архітектури [5].

Ознайомившись із публікаціями, що з'являлися з часу виникнення цієї парадигми, можна побачити, що аналогічні дискусії вже мали місце раніше, і за весь період її існування світ так і не дійшов єдиного висновку щодо того, чи варто використовувати реактивне програмування, чи слід від нього відмовитися. Це може свідчити про те, що ця технологія має такі ж самі шанси на існування, як і імперативна, і за відповідних умов здатна проявити себе неперевершено [6].

Мета дослідження

Метою статті є аналіз можливостей та доцільності застосування реактивного програмування у розробці сучасних вебсервісів. Визначити переваги та недоліки реактивної парадигми порівняно з традиційними імперативними підходами, а також окреслити умови, за яких використання реактивного програмування є найбільш ефективним.

https://doi.org/10.32782/mathematical-modelling/2026-9-1-36 ISSN 2618-0332 (print), 2618-0340 (online)

Виклад основного матеріалу дослідження

1. Сучасні тенденції та проблеми вебсистем

За останні шість років напрямок розробки вебзастосунків змістився у бік розподілених систем, де незалежні компоненти взаємодіють між собою через запити один до одного. Такий варіант архітектури, відомий як мікросервісна архітектура, швидко замінив більшість монолітних структур. Завдяки цій зміні сучасні сервіси повинні обробляти значно більше запитів, адже зараз інформацію можуть запитувати не лише користувачі, а й інші сервіси. Час відгуку став критично важливим, оскільки затримки у відповіді можуть призвести до втрати користувачів і негативно вплинути на подальший розвиток бізнесу. Проблема масштабованості залишається актуальною: система повинна справлятися зі значними коливаннями трафіку, який може варіюватися від сотень до тисяч одночасних користувачів у різні періоди.

Традиційні підходи до програмування часто стикаються зі складнощами асинхронності, конкурентності та збоїв у розподілених середовищах. Класичні блокуючі моделі значно обмежують кількість одночасних з'єднань, які система може підтримувати без збільшення кількості потоків, а витрати на перемикання контексту потоків зростають із збільшенням кількості одночасних користувачів. Реактивне програмування має ресурси для вирішення цієї проблеми, оскільки використання даної технології дозволяє забезпечити значне підвищення пропускної здатності та зниження затримки у порівнянні з традиційними підходами, особливо при обробці навантажень із змінною мережевою затримкою [7].

2. Основні принципи Реактивного програмування

Реактивне програмування – це декларативна парадигма, що фокусується на обробці асинхронних потоків даних та поширенні змін у системі. У реактивних системах компоненти реагують на дані в реальному часі, подібно до того, як електронна таблиця автоматично оновлює значення клітинок при зміні вихідних даних. Практика показує, що реактивні реалізації здатні суттєво зменшити затримку відповіді та підвищити пропускну здатність у порівнянні з традиційними блокуючими моделями, особливо при обробці великої кількості одночасних з'єднань та змінних навантажень.

2.1. Асинхронність та неблокуючі операції

Асинхронність і неблокуючі операції є фундаментальними характеристиками реактивного програмування. У традиційних (імперативних) системах виконання коду зазвичай відбувається послідовно: кожна операція виконується одна за одною, і якщо якась із них потребує часу (наприклад, очікування відповіді від бази даних чи зовнішнього сервісу), потік виконання блокується до завершення цієї операції. Це означає, що ресурси (наприклад, потоки процесора) залишаються неактивними, поки очікується результат, що обмежує масштабованість і ефективність системи.

У реактивному підході обробка даних організована як потік подій, де кожна операція може виконуватися незалежно від інших і не блокує основний потік виконання. Коли система стикається з операцією, яка потребує часу, наприклад, запит до бази даних або стороннього сервісу, вона не зупиняє виконання, а переходить до обробки інших подій або запитів. Коли результат стає доступним, система «реагує» на цю подію – наприклад, викликає відповідний обробник або оновлює інтерфейс користувача.

Цей підхід дозволяє ефективніше використовувати апаратні ресурси, оскільки потоки не просто очікують завершення операцій, а виконують інші завдання. У результаті реактивні системи здатні обробляти значно більшу кількість одночасних запитів без необхідності створювати велику кількість потоків, що особливо важливо для високонавантажених або розподілених систем.

Відмінність від імперативного підходу полягає у тому, що в імперативних системах розробник явно керує порядком виконання операцій і часто стикається з проблемами блокування, тоді як у реактивних системах логіка побудована навколо реакції на події та зміни стану, що дозволяє уникати блокувань і підвищує гнучкість архітектури.

https://doi.org/10.32782/mathematical-modelling/2026-9-1-36 ISSN 2618-0332 (print), 2618-0340 (online)

2.2. Декларативний стиль програмування

Декларативний стиль програмування є фундаментальною рисою реактивних систем. Тут розробник описує бажану поведінку системи у відповідь на потоки даних або події, без прив'язки до послідовності подій. Це дозволяє зосередитися на тому, що саме має відбутися, а не на деталях реалізації процесу.

Завдяки такому стилю компоненти системи стають менш залежними один від одного. Аналіз архітектурної складності показує, що системи, побудовані на декларативних принципах, мають менше надлишкового мережевого трафіку між сервісами під час пікових навантажень, а також демонструють кращу стійкість до локалізованих збоїв.

У реактивних системах, побудованих на декларативному підході, локальні відмови впливають лише на обмежену частину функціональності, тоді як у жорстко зв'язаних архітектурах наслідки збоїв можуть бути значно масштабнішими. Це підвищує загальну надійність і спрощує підтримку великих розподілених систем.

2.3. Backpressure management

Backpressure management – це ключовий механізм контролю потоку даних у реактивних системах, який забезпечує стабільність та ефективність роботи під час високих навантажень. Його суть полягає в тому, що споживач даних може сигналізувати про необхідність уповільнення або тимчасового призупинення потоку, якщо обробка даних не встигає за їх надходженням. Це дозволяє уникати перевантаження системи, втрат даних та переповнення буферів.

У розподілених середовищах, де навантаження може змінюватися динамічно, backpressure management стає особливо важливим. Системи, які реалізують динамічний backpressure, здатні підтримувати стабільне споживання пам'яті навіть при різких стрибках трафіку, а також запобігати втраті пакетів даних.

Саме завдяки цьому підходу реактивні системи можуть ефективно адаптуватися до змінних умов роботи, забезпечуючи стабільність і надійність навіть у ситуаціях, коли навантаження перевищує базовий рівень у кілька разів. Як вже було написано вище це особливо важливо для сучасних вебсервісів, які повинні обробляти тисячі одночасних запитів та гарантувати якість обслуговування користувачів незалежно від зовнішніх факторів.

2.4. Подієво-орієнтована архітектура (Event driven architecture)

У реактивних системах компоненти взаємодіють між собою через події, а не через прямі виклики методів. Це означає, що кожен компонент реагує на події у міру їх надходження, а не ініціює взаємодію безпосередньо. Такий підхід забезпечує слабке зв'язування (loose coupling) між компонентами, що значно підвищує гнучкість архітектури та спрощує її масштабування.

Подієво-орієнтована архітектура дозволяє системі реагувати на події у реальному часі, що є критично важливим для розподілених і високонавантажених середовищ. Крім того, подієво-орієнтований підхід сприяє зменшенню кількості залежностей між компонентами, що спрощує підтримку та розвиток системи. Компоненти можуть розвиватися незалежно один від одного, а нові функціональні можливості можуть додаватися без ризику порушення роботи існуючих частин системи. Це особливо важливо для сучасних вебсервісів, які повинні залишатися гнучкими та стійкими до змін у динамічному середовищі.

3. Реактивні фреймворки та бібліотеки

Існує низка популярних бібліотек і фреймворків, які спрощують впровадження реактивних систем у різних мовах програмування, особливо в Java та Scala. Еволюція цих технологій була зумовлена зростаючими вимогами сучасних розподілених застосунків. Аналіз продуктивності різних фреймворків демонструє стабільне зростання ефективності, хоча підходи до реалізації та оптимізації можуть відрізнятися. Дослідження безпеки реактивних систем показують скорочення вікна вразливості на 28 % завдяки зменшенню часу між виявленням загрози та реакцією на неї, що є критично важливим для сучасних хмарних моделей безпеки [8].

https://doi.org/10.32782/mathematical-modelling/2026-9-1-36 ISSN 2618-0332 (print), 2618-0340 (online)

Project Reactor – це бібліотека для реактивного програмування на JVM, яка реалізує стандарт Reactive Streams. Вона надає два основних типи: Flux (для роботи з потоками з багатьма елементами) та Mono (для обробки одного елемента або його відсутності). Project Reactor лежить в основі Spring WebFlux і дозволяє будувати неблокуючі, масштабовані вебсервіси. Бібліотека підтримує оператори для трансформації, фільтрації, комбінування та агрегації потоків даних, а також має вбудовані засоби для обробки помилок і управління backpressure. Project Reactor використовується у великих корпоративних рішеннях, зокрема у Netflix, де потрібна обробка великої кількості одночасних запитів у реальному часі.

RxJava – одна з найпопулярніших бібліотек для реактивного програмування на Java, яка реалізує концепції ReactiveX. Вона дозволяє створювати асинхронні та подієво-орієнтовані програми на основі Observable-послідовностей. RxJava підтримує багатий набір операторів для роботи з потоками, планувальники для керування багатопоточністю, а також інтеграцію з іншими бібліотеками та фреймворками. Бібліотека активно використовується у мобільних застосунках (наприклад, Trello для Android), а також у фінансових і аналітичних системах, де важлива швидка реакція на події та зміни даних.

Akka Streams – компонент фреймворку Akka, орієнтований на обробку потоків даних із підтримкою backpressure. Akka Streams побудований на акторній моделі(actor model), що дозволяє створювати розподілені, стійкі до збоїв системи з високою пропускною здатністю. Бібліотека надає декларативний API для побудови складних графів обробки даних, підтримує інтеграцію з різними джерелами та приймачами даних (наприклад, файли, мережа, бази даних). Akka Streams часто використовується у телекомунікаційних платформах, системах моніторингу, аналітики та IoT-рішеннях, де необхідна обробка великих обсягів даних у реальному часі.

Spring WebFlux – це реактивний вебфреймворк у складі Spring, який дозволяє створювати неблокуючі вебзастосунки з ефективною обробкою асинхронних запитів і відповідей. WebFlux побудований на основі Project Reactor і підтримує як анотаційний, так і функціональний стиль програмування. Фреймворк інтегрується з іншими компонентами Spring, забезпечує підтримку WebSocket, SSE (Server-Sent Events) та REST API. WebFlux використовується у мікросервісних архітектурах, сервісах потокового відео, онлайн-магазинах та фінансових платформах, де важливо забезпечити швидку реакцію на запити та стабільну роботу під час пікових навантажень.

RSocket – це сучасний мережевий протокол, спеціально розроблений для реактивних систем. RSocket підтримує асинхронну двосторонню взаємодію між клієнтом і сервером, дозволяючи реалізовувати різні моделі обміну даними: запит-відповідь, потік даних, fire-and-forget та канал (двосторонній стрімінг). Протокол оптимізований для роботи у розподілених середовищах, забезпечує низькі затримки, ефективне використання мережевих ресурсів і вбудовану підтримку backpressure на транспортному рівні. RSocket активно використовується у високонавантажених сервісах, де важлива швидка реакція на події, наприклад, у фінансових платформах, ігрових сервісах та системах реального часу. Його інтеграція з Project Reactor і Spring WebFlux дозволяє легко впроваджувати реактивні мережеві взаємодії у сучасних Java сервісах.

Ці бібліотеки та фреймворки надають розробникам інструменти для побудови масштабованих, гнучких і стійких до збоїв систем. Вони активно використовуються у великих компаніях і стартапах, які прагнуть забезпечити високу якість обслуговування користувачів, ефективне використання ресурсів та стабільність роботи навіть у складних розподілених середовищах.

4. Переваги реактивного програмування у розподілених системах

Впровадження реактивного програмування у розподілених системах надає низку суттєвих переваг, які важливо враховувати при виборі архітектурних рішень для сучасних цифрових продуктів.

4.1. Підвищення продуктивності та стабільності

Реактивні системи демонструють значно кращу продуктивність у сценаріях з високою конкуренцією, особливо при обробці великої кількості одночасних з’єднань. На практиці такі

https://doi.org/10.32782/mathematical-modelling/2026-9-1-36 ISSN 2618-0332 (print), 2618-0340 (online)

системи здатні підтримувати стабільний рівень пропускної здатності навіть при суттєвому зростанні навантаження, тоді як традиційні підходи часто стикаються з різким падінням ефективності. Це безпосередньо впливає на користувацький досвід: середній час завантаження сторінки скорочується, а система залишається стабільною навіть під час пікових навантажень.

4.2. Відгуковість і передбачувана затримка

Асинхронна природа реактивних систем забезпечує швидку реакцію на запити користувачів і події, навіть за умов змінного навантаження. Реактивні реалізації дозволяють підтримувати низькі затримки відповіді у більшості сценаріїв, що особливо важливо для застосунків із жорсткими вимогами до часу відгуку. Зростання обсягу оброблюваних даних або кількості повідомлень у системі не призводить до різкого погіршення продуктивності, що гарантує стабільність роботи сервісу.

4.3. Ефективне використання ресурсів

Реактивні системи дозволяють ефективніше використовувати апаратні ресурси, зменшуючи потребу у великій кількості потоків і оперативної пам'яті. Завдяки неблокуючій обробці запитів, такі системи можуть обслуговувати більше одночасних підключень на тих самих ресурсах, що знижує інфраструктурні витрати та спрощує масштабування. Зростання кількості підключень не призводить до експоненційного збільшення споживання пам'яті, що особливо важливо для високонавантажених сервісів.

4.4. Гнучкість у роботі з потоками даних

Реактивне програмування ідеально підходить для побудови систем, які обробляють події та потоки даних у реальному часі. Можливість легко комбінувати, трансформувати та обробляти потоки дозволяє створювати ефективні рішення для обробки великих обсягів інформації, наприклад, у фінансових платформах, системах моніторингу чи аналітики.

4.5. Стійкість до збоїв і самовідновлення

Асинхронна передача повідомлень і використання акторної моделі(actor model) сприяють підвищенню стійкості системи до збоїв. Реактивні системи здатні ізолювати помилки, локалізувати їх у межах окремих компонентів і запобігати поширенню збоїв на всю систему. Використання шаблонів по типу circuit breaker дозволяє швидко відновлюватися після часткових відмов і підтримувати працездатність навіть при виході з ладу окремих елементів.

4.6. Зниження операційних витрат і покращення підтримки

Завдяки ізоляції збоїв і самовідновленню, реактивні системи потребують менше часу на розслідування та усунення інцидентів. Команди, які працюють із такими системами, витрачають менше ресурсів на підтримку інфраструктури, а кількість інцидентів, що впливають на роботу сервісу, зменшується. Крім того, реактивні системи генерують менше хибних сповіщень про помилки, що дозволяє зосередитися на реальних проблемах і підвищує ефективність роботи команди підтримки.

5. Виклики та обмеження реактивного програмування

Розглянувши переваги наведені вище постає питання, чому тоді всі масово не переходять на цю технологію, якщо вона настільки покращує продуктивність і стабільність сервісів.

5.1. Крива навчання та адаптації

Перехід до реактивного підходу часто супроводжується тимчасовим зниженням продуктивності команд розробників, особливо у перші 2–3 місяці. Це пов'язано з необхідністю освоїти нові патерни, концепції та інструменти, які суттєво відрізняються від традиційних імперативних моделей. Найбільше труднощів виникає у розробників із великим досвідом роботи з класичними підходами, оскільки реактивна логіка потребує іншого стилю мислення. Важливу роль відіграє якісне навчання та структурований процес передачі знань, що дозволить значно скоротити період адаптації та зменшити негативний вплив на продуктивність, але потребує вкладу часу та грошей від бізнесу.

https://doi.org/10.32782/mathematical-modelling/2026-9-1-36 ISSN 2618-0332 (print), 2618-0340 (online)

5.2. Складність коду та підтримки

Реактивні реалізації часто мають більш складну структуру, особливо у частині композиції та трансформації потоків даних. Така структура при неправильності імплементації через брак досвіду може виглядати складною для розуміння, особливо через низьку зв'язаність. Це призводить до збільшення складності коду, що негативно впливає на його підтримку, тестування та розвиток. Особливу увагу слід приділяти управлінню підписками та очищенню ресурсів, оскільки неправильна робота з цими аспектами може призвести до проблем із пам'яттю та витоками ресурсів.

5.3. Відлагодження та діагностика

Відлагодження асинхронного коду є більш складним порівняно з синхронними реалізаціями. Потік виконання стає менш очевидним, а stack trace часто містить менше корисної інформації для пошуку помилок. Найбільше часу розробники витрачають на діагностику проблем, пов'язаних із обробкою backpressure та контролем конкурентності. Значна частина помилок у реактивних системах виникає через неправильні припущення щодо порядку виконання або моменту обробки подій.

5.4. Підвищені вимоги до спостережуваності (observability)

Для ефективного моніторингу та діагностики реактивних систем потрібно більше точок інструментування у потоці запитів. Якщо в стандартних системах відслідкувати результат та в якому моменті він видав помилку досить просто, в реактивних системах необхідно вводити додаткові міри боротьби з цим. Недостатній моніторинг може призвести до «сліпих зон» у системі, де проблеми залишаються непоміченими.

5.5. Архітектурна неоднорідність та гібридні підходи

Під час переходу до реактивної парадигми часто виникають гібридні рішення, де частина системи працює за реактивними принципами, а частина – за традиційними. Це може бути великою проблемою, коли одній системі потрібно взаємодіяти з іншою, тому цього краще уникати і розділяти такі проекти на різні сервіси якщо це можливо.

5.6. Підтримка та масштабування знань

На початковому етапі впровадження реактивного програмування кодова база може бути менш зрозумілою та складнішою для підтримки. Однак із набуттям досвіду команда поступово вирівнює рівень підтримки та розуміння системи, і ці труднощі стають менш помітними.

6. Коли доцільно використовувати реактивне програмування

Використання реактивного програмування найбільш виправдане у тих випадках, коли система повинна обробляти велику кількість одночасних підключень або працювати під високим навантаженням. Це стосується, зокрема, платформ потокового відео, онлайн-ігор, фінансових сервісів чи великих маркетплейсів, де неблокуюча обробка подій дозволяє ефективно використовувати ресурси та підтримувати стабільний час відповіді навіть у пікові періоди.

Реактивний підхід також є оптимальним вибором для застосунків, які інтенсивно взаємодіють із мережею або зовнішніми сервісами, наприклад, у мікросервісних архітектурах. У таких системах велика кількість внутрішніх і зовнішніх запитів може призвести до перевантаження традиційних моделей, тоді як реактивна обробка дозволяє уникати блокування потоків під час очікування відповіді.

Особливо ефективним реактивне програмування є для систем реального часу та обробки потоків подій, таких як системи моніторингу, аналітики, IoT-рішення чи біржові платформи. Можливість легко комбінувати, фільтрувати та трансформувати потоки даних дозволяє створювати гнучкі та масштабовані рішення для обробки великих обсягів інформації.

Ще однією важливою перевагою реактивного підходу є підвищення стійкості системи до збоїв. Завдяки ізоляції помилок, використанню шаблонів по типу circuit breaker, система може залишатися працездатною навіть при часткових відмовах окремих компонентів, що запобігає каскадним збоям.

https://doi.org/10.32782/mathematical-modelling/2026-9-1-36 ISSN 2618-0332 (print), 2618-0340 (online)

Реактивні системи добре адаптуються до змінного або непередбачуваного навантаження, автоматично масштабуючи ресурси та підтримуючи стабільну роботу без необхідності ручного втручання. Водночас, для простих CRUD-застосунків, невеликих внутрішніх інструментів або систем із низьким навантаженням реактивне програмування може бути надмірним і ускладнювати розробку. У таких випадках традиційні імперативні підходи залишаються більш простими та ефективними.

Варто зазначити, що підвищена продуктивність, яку забезпечує реактивне програмування, не завжди є критичною для більшості сервісів. У багатьох випадках достатньо використовувати прості та перевірені архітектурні підходи, які легше впроваджувати та підтримувати. Реактивні системи демонструють себе найкраще в середовищах з високим навантаженням, де необхідна масштабованість і стійкість до збоїв. Якщо система працює у стабільному режимі з невеликою кількістю одночасних запитів, переваги реактивної парадигми можуть бути незначними або навіть нівелюватися.

Висновки

Реактивне програмування пропонує сучасний набір принципів і інструментів для побудови надійних, масштабованих і ефективних розподілених систем. Завдяки асинхронності, подієво-орієнтованій архітектурі та управлінню backpressure, ця парадигма дозволяє вирішувати ключові проблеми, які виникають при розробці вебсервісів у складних і динамічних середовищах. Переваги реактивного підходу проявляються у підвищенні продуктивності, стійкості до збоїв, гнучкості та ефективному використанні ресурсів, що особливо важливо для систем із високим навантаженням, великою кількістю одночасних підключень та вимогам до обробки подій у реальному часі.

Водночас, впровадження реактивного програмування супроводжується низкою викликів, такі як складність освоєння нових концепцій, підвищені вимоги до моніторингу та діагностики, а також необхідність ретельного ресурсами при їх обробці. Перехід до реактивної парадигми потребує часу для адаптації команди, структурованого навчання та зміни підходів до розробки й підтримки системи. Особливо важливо враховувати, що продуктивність і масштабованість, які забезпечує реактивний підхід, не завжди є критично необхідними для більшості сервісів. У багатьох випадках прості та перевірені імперативні архітектури залишаються більш доцільними, особливо для невеликих або стабільних систем із низьким навантаженням.

Аналіз сучасних досліджень і практичних кейсів показує, що реактивне програмування не є універсальним рішенням, а скоріше потужним інструментом для тих проєктів, які стикаються з високими вимогами до продуктивності, стійкості та гнучкості. Вибір цієї парадигми має базуватися на реальних потребах проєкту, характері навантаження, вимогах до масштабування та стійкості, а також готовності команди до освоєння нових підходів.

З огляду на постійне зростання складності та масштабів сучасних розподілених систем, принципи реактивного програмування, ймовірно, стануть ще більш актуальними у майбутньому. Вони можуть забезпечити фундамент для створення наступного покоління вебсервісів, які відповідатимуть вимогам цифрової епохи щодо продуктивності, стійкості та масштабованості. Проте, як показує аналіз публікацій і практики, остаточний вибір між реактивною та імперативною парадигмою повинен бути зваженим і враховувати специфіку кожного проєкту.

Список використаної літератури

1. Why Reactive Might Be Dead: Spring Boot + Java 21 Virtual Threads Are All You Need. *Medium*. URL: https://medium.com/@kanhaaggarwal/why-reactive-might-be-dead-spring-boot-java-21-virtual-threads-are-all-you-need-2d6d545fc18b (дата звернення: 25.03.2026).

2. Höjvall M. The Reactive Java era is over. Here is why. *Medium*. URL: https://medium.com/alphadev-thoughts/the-reactive-java-era-is-over-here-is-why-5885caacdf43 (дата звернення: 25.03.2026).
3. Yanev I. Why we discarded Reactive systems architecture from our code?. *Dev.to*. URL: https://dev.to/yanev/why-we-discarded-reactive-systems-architecture-from-our-code-19ni (дата звернення: 26.03.2026).
4. The Shift Toward Reactive Programming in Modern Web Development. *Dev.to*. URL: https://dev.to/softwaredeveloperhub01/the-shift-toward-reactive-programming-in-modern-web-development-10dd (дата звернення: 23.03.2026).
5. Микитин А. Реактивне програмування на Spring Boot: мій досвід, приклади та розбір загальноприйнятих підходів. *DOU*. URL: https://dou.ua/forums/topic/53998/ (дата звернення: 24.03.2026).
6. Purnomo J. Reactive vs Imperative Programming: Choosing the Right Paradigm for Your Project. *Medium*. URL: https://medium.com/@jonatanlaksamanapurnomo/reactive-vs-imperative-programming-3c6a0d267a3a (дата звернення: 23.03.2026).
7. Wojciech O. Reactive vs imperative – performance. *Medium*. URL: https://medium.com/@w_olech/reactive-vs-imperative-performance-752bd79f24c (дата звернення: 27.03.2026).
8. Chaurasia B, Verma A. A Comprehensive Study on Failure Detectors of Distributed Systems. *Journal of Scientific Research*. 2020. Vol. 64. P. 250–260. DOI: https://doi.org/10.37398/JSR.2020.640235
9. Hyseni D, Piraj N, Shabani I. The Use of Reactive Programming in the Proposed Model for Cloud Security Controlled by ITSS. *Computers*, 2022. Vol. 11. № 5. 62. DOI: https://doi.org/10.3390/computers11050062

References

1. Why Reactive Might Be Dead: Spring Boot + Java 21 Virtual Threads Are All You Need. (2024). *Medium*. https://medium.com/@kanhaaggarwal/why-reactive-might-be-dead-spring-boot-java-21-virtual-threads-are-all-you-need-2d6d545fc18b [in English].
2. Höjvall, M. (2024). The Reactive Java era is over. *Medium*. https://medium.com/alphadev-thoughts/the-reactive-java-era-is-over-here-is-why-5885caacdf43 [in English].
3. Yanev, I. (2024). Why we discarded Reactive systems architecture from our code? *Dev.to*. https://dev.to/yanev/why-we-discarded-reactive-systems-architecture-from-our-code-19ni [in English].
4. The Shift Toward Reactive Programming in Modern Web Development. (2025). *Dev.to*. https://dev.to/softwaredeveloperhub01/the-shift-toward-reactive-programming-in-modern-web-development-10dd [in English].
5. Mykytyn, A. (2025). Reaktyvne prohramuvannia na Spring Boot: mii dosvid, pryklady ta rozbir zahalnopryiniatykh pidkhodiv. [Reactive programming on Spring Boot: My experience, examples, and analysis of generally accepted approaches]. *DOU*. https://dou.ua/forums/topic/53998/ [in Ukrainian].
6. Purnomo, J. (2025). Reactive vs Imperative Programming: Choosing the Right Paradigm for Your Project. *Medium*. https://medium.com/@jonatanlaksamanapurnomo/reactive-vs-imperative-programming-3c6a0d267a3a [in English].
7. Wojciech, O. (2019). Reactive vs imperative – performance. *Medium*. https://medium.com/@w_olech/reactive-vs-imperative-performance-752bd79f24c [in English].
8. Chaurasia, B., & Verma, A. (2020). A Comprehensive Study on Failure Detectors of Distributed Systems. *Journal of scientific research*, *64*(02), 250–260. DOI: https://doi.org/10.37398/jsr.2020.640235 [in English].

9. Hyseni, D., Piraj, N., Çiço, B., & Shabani, I. (2022). The Use of Reactive Programming in the Proposed Model for Cloud Security Controlled by ITSS. *Computers*, *11* (5), 62. DOI: https://doi.org/10.3390/computers11050062 [in English].

Труфанов Нікіта Іванович – магістр кафедри комп'ютерної інженерії та програмування Національного технічного університету «Харківський політехнічний інститут». E-mail: nikita.i.trufanov@gmail.com, ORCID: 0009-0009-3729-6400.

Поворознюк Анатолій Іванович – д.т.н., професор, професор кафедри комп'ютерної інженерії та програмування Національного технічного університету «Харківський політехнічний інститут». E-mail: ai.povoroznjuk@gmail.com, ORCID: 0000-0003-2499-2350.

Trufanov Nikita Ivanovych – Master's Student at the Department of Computer Engineering and Programming of the National Technical University "Kharkiv Polytechnic Institute". E-mail: nikita.i.trufanov@gmail.com, ORCID: 0009-0009-3729-6400.

Povoroznyuk Anatolii Ivanovych – Doctor of Technical Sciences, Professor, Professor at the Department of Computer Engineering and Programming of the National Technical University "Kharkiv Polytechnic Institute". E-mail: ai.povoroznjuk@gmail.com, ORCID: 0000-0003-2499-2350.

Дата першого надходження статті до видання: 30.03.2026
Дата прийняття статті до друку після рецензування: 08.05.2026
Дата публікації (оприлюднення) статті: 01.07.2026