

ІНСТИТУТ ПРОБЛЕМ МОДЕЛЮВАННЯ В ЕНЕРГЕТИЦІ ІМ. Г.Є. ПУХОВА
НАЦІОНАЛЬНА АКАДЕМІЯ НАУК УКРАЇНИ
ІНСТИТУТ ПРОБЛЕМ МОДЕЛЮВАННЯ В ЕНЕРГЕТИЦІ ІМ. Г.Є. ПУХОВА
НАЦІОНАЛЬНА АКАДЕМІЯ НАУК УКРАЇНИ

Кваліфікаційна наукова праця
на правах рукопису

ЯРОШИНСЬКИЙ МИКОЛА СЕРГІЙОВИЧ

УДК 681.5:004.75:004.4`6:004.94

ДИСЕРТАЦІЯ
МЕТОД ВИЯВЛЕННЯ НЕСУМІСНИХ ВЕРСІЙ СЕРВІСІВ ПЕРЕД
РОЗГОРТАННЯМ РОЗПОДІЛЕНИХ ПРОГРАМНИХ СИСТЕМ

122 Комп'ютерні науки
12 Інформаційні технології

Подається на здобуття наукового ступеня доктор філософії

Дисертація містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

 М.С. Ярошинський

Науковий керівник: Мохор Володимир Володимирович, член-кореспондент
НАН України, доктор технічних наук, професор

Київ — 2025

АНОТАЦІЯ

Ярошинський М.С. Метод виявлення несумісних версій сервісів перед розгортанням розподілених програмних систем.

Дисертація на здобуття наукового ступеня доктора філософії за спеціальністю 122 – Комп’ютерні науки. – Інститут проблем моделювання в енергетиці ім. Г.Є. Пухова НАН України, Київ, 2025.

У дисертації розроблено метод виявлення несумісних версій сервісів перед розгортанням розподілених програмних систем. Обґрунтовано актуальність задачі, пов’язаної з асинхронними змінами прикладних програмних інтерфейсів (API), які можуть призводити до зниження надійності систем. Запропонований метод базується на формалізованих критеріях сумісності API та забезпечує автоматизоване блокування несумісних релізів, що дозволяє запобігти появі несумісних змін у робочому середовищі.

Мета роботи полягає у розробці та обґрунтуванні методу виявлення несумісних версій сервісів перед розгортанням розподілених програмних систем. Цей метод має забезпечити раннє виявлення конфліктів у міжсервісній взаємодії, що виникають внаслідок асинхронних змін в їхніх інтерфейсах. Запропонований підхід має на меті підвищити надійність та стабільність функціонування системи в цілому, запобігаючи збоям, що можуть бути спричинені несумісностями API, та мінімізуючи ризики під час її еволюційного розвитку.

Об’єктом дослідження є виявлення несумісних версій сервісів перед розгортанням розподілених програмних систем з урахуванням процесів еволюції API.

Предмет дослідження – метод виявлення несумісних версій сервісів перед розгортанням розподілених програмних систем.

Наукова новизна роботи, що виносяться на захист, характеризується наступними твердженнями:

1. Вперше розроблено метод виявлення несумісних версій сервісів перед розгортанням розподілених програмних систем, який, на відміну від існуючих, базується на формалізованих критеріях сумісності API, що забезпечує автоматизацію блокування несумісних релізів, знижує ризики каскадних відмов через несумісні зміни API і гарантує вибір сумісних конфігурацій не допускаючи появи несумісних API в робочому середовищі.

2. Розроблено програмне забезпечення для перевірки ефективності методу, що дозволило здійснити його експериментальну апробацію на архітектурно-програмному стенді та підтвердити придатність до практичного застосування у реальних сценаріях розподілених систем. Реалізовані інструменти забезпечують автоматизовану перевірку сумісності gRPC- та JSON Schema-сервісів, інтегруються у конвеєри CI/CD та перемикання трафіку на новий реліз, що дозволило обґрунтувати підвищення надійності і відмовостійкості при оновленні критично важливих інформаційних сервісів.

3. Дістали подальшого розвитку існуючі стратегії мінімізації впливу у розподілених системах за рахунок їх комбінації із запропонованим методом, що дозволило обґрунтувати поєднання з процесуальним підходом. Така інтеграція забезпечує багаторівневу перевірку стабільності: на першому рівні відбувається автоматизоване блокування несумісних версій API ще до моменту їхнього потрапляння у тестове середовище, а на другому — здійснюється відтворення повноцінних сценаріїв роботи системи в умовах тестового середовища. Це дозволяє не лише гарантувати технічну сумісність інтерфейсів, але й оцінити поведінку сервісів під навантаженнями, наближеними до продуктивних.

У роботі проаналізовано сучасні підходи до управління еволюцією програмних інтерфейсів (API) у розподілених системах, включаючи стратегії мінімізації впливу несумісних змін та методи забезпечення сумісності між сервісами. Проведено систематизацію причин виникнення несумісностей, розкрито їхній вплив на працездатність розподілених систем і виявлено обмеження існуючих практик DevOps та CI/CD, які не гарантують раннього виявлення критичних відмов.

У роботі розкрито поняття несумісних змін API та обґрунтовано метод виявлення несумісних версій сервісів перед розгортанням. Запропоновано застосування «Реєстру сумісності API», який інтегрується з конвеєрами CI/CD, автоматично перевіряє залежності між сервісами та формує сценарії розгортання лише для сумісних конфігурацій. Це дало змогу створити механізм проактивного контролю сумісності, що усуває проблему появи каскадних відмов у робочому середовищі.

У вступі обґрунтовано актуальність теми, сформульовано мету, поставлено завдання, визначено предмет, об'єкт та методи дослідження, викладено основні положення наукової новизни та практичного значення одержаних результатів.

У першому розділі розглянуто природу та причини еволюції API, зокрема розширення функціональності, міграцію технологічної бази, оптимізацію продуктивності та оновлення дизайну. Значна увага приділена поняттю «несумісних змін» та викликам, які вони створюють у складних розподілених системах. Аналізується їх вплив на працездатність клієнтських застосунків і всю екосистему сервісів. Формалізується модель каскадних відмов у розподілених програмних системах при появі несумісних змін.

Систематизуються стратегії мінімізації ризиків: staging-тестування, автоматична генерація схем протоколів, використання форматів з адаптивною

сумісністю, стратегія Blue/Green-розгортання та включення версій API у протокол.

У другому розділі проведено системний аналіз архітектур розподілених програмних систем, що підтвердив різноманітність підходів до організації взаємодії сервісів. Це дало змогу сформулювати концептуальне підґрунтя для побудови моделі середовища, у якому виникають ризики несумісності API. Встановлено, що при сучасних процесах розгортання в різних середовищах інтеграція методу виявлення несумісних версій сервісів перед розгортанням розподілених програмних систем дозволить автоматично ідентифікувати конфлікти між версіями сервісів, запобігаючи утворенню некоректних конфігурацій у робочому середовищі.

У межах дослідження розглянуто різні варіанти середовищ для реалізації методу та обґрунтовано вибір Kubernetes як оптимального рішення. Його оркестраційні можливості забезпечують масштабованість, контроль життєвого циклу сервісів і гнучке керування оновленнями, що дозволяє ефективно інтегрувати запропонований метод у процеси розгортання. Таким чином, розділ закладає науково-методичну основу для подальшої програмної реалізації розробленого методу та експериментальної перевірки його ефективності.

У третьому розділі було реалізовано архітектурний та програмний підхід до створення експериментального стенду, що забезпечує відтворення умов реального функціонування розподілених систем. Розроблений метод було включено в себе спеціалізовану аплікацію, яка дозволяє автоматизувати процес виявлення несумісних версій API ще до моменту розгортання. Це дало змогу перевірити не лише сумісність на рівні окремих сервісів, але й коректність функціонування усієї розподіленої програмної системи в умовах каскадних залежностей. Побудована архітектура стенду довела свою придатність як для

імітації робочих сценаріїв, так і для відтворення контрольованих відмов, що є передумовою для верифікації запропонованого методу.

Дістали подальшого розвитку існуючі стратегії мінімізації впливу несумісних змін API у розподілених системах за рахунок їх комбінації із запропонованим методом, а саме поєднання описаного методу з процесуальним підходом.

Четвертий розділ описує проведене експериментальне дослідження, яке підтвердило ефективність запропонованого методу виявлення несумісних версій сервісів перед розгортанням. Було реалізовано інструментальну підтримку для перевірки сумісності gRPC- та JSON Schema-сервісів. У процесі контрольованого внесення несумісних змін було показано, що система здатна своєчасно блокувати небезпечні релізи та запобігати каскадним відмовам. Використання стратегії перемикавання трафіку дозволило оцінити робастність методу, а сценарії скоординованого оновлення сервісів довели його придатність для систем із високою частотою змін. Результати експериментів засвідчили, що інтеграція запропонованого підходу у конвеєри CI/CD підвищує рівень надійності та стабільності розподілених програмних систем, знижуючи ризики збоїв на етапі еволюції API, які призводять до появи несумісних змін при міжсервісній комунікації.

У висновках наведено отримані наукові та практичні результати дослідження.

Актуальність роботи. Сучасні програмні системи, що лежать в основі критично важливих застосунків та комерційних продуктів, дедалі частіше реалізуються у формі розподілених архітектур. Такі підходи забезпечують масштабованість, відмовостійкість та гнучкість розробки, проте водночас створюють нові виклики, пов'язані з узгодженням численних компонентів. мінімізації впливу несумісних змін API у розподілених системах стає особливо

гострою у системах управління критичною інфраструктурою, де безперервність функціонування є визначальною умовою надійності.

Еволюція програмних систем неминуче супроводжується змінами у прикладних програмних інтерфейсах. В умовах асинхронного оновлення сервісів це призводить до ризику несумісності їхніх версій, коли очікування одного компонента не відповідають форматам або протоколам іншого. Такі несумісні зміни можуть спричинити каскадні відмови, що унеможлиблює стабільну роботу всієї системи.

Таким чином, актуальність роботи визначається необхідністю створення формалізованих методів і механізмів, здатних забезпечити контроль сумісності API ще до моменту розгортання. Це дозволяє підвищити надійність та передбачуваність еволюції програмних систем, зокрема у сферах, де збій навіть одного компонента може призвести до значних економічних втрат або загроз безпеці.

Практична значимість результатів дисертаційної роботи полягає у створенні методу, що дозволяє інженерам і командам розробки забезпечувати підвищений рівень надійності та стабільності розподілених програмних систем. Застосування методу на етапі підготовки до розгортання уможливлює автоматичне виявлення та усунення несумісностей між сервісами ще до того, як вони можуть спричинити збої у роботі. Це безпосередньо знижує кількість інцидентів, пов'язаних із міжсервісними конфліктами, і скорочує витрати часу на їх діагностику та усунення.

Запропонований метод інтегрується у сучасні DevOps-практики, зокрема у конвеєри безперервної інтеграції та доставки (CI/CD). Це надає додатковий рівень автоматизованого контролю якості та підтримує безперервність функціонування сервісів, що є особливо важливим для критично важливих застосувань. Використання стратегії перемикання трафіку

разом із перевіркою сумісності API мінімізує ризики появи каскадних відмов під час оновлення, підвищуючи надійність системної інтеграції.

Результати дослідження можуть бути впроваджені у комерційних та державних організаціях, які експлуатують масштабні розподілені програмні системи. Це стосується насамперед галузей, де стабільність і передбачуваність роботи є критичною умовою: енергетики, транспорту, телекомунікацій, авіаційних і фінансових систем. Таким чином, практична цінність полягає не лише у підвищенні відмовостійкості, але й у забезпеченні економічного ефекту завдяки зменшенню витрат на усунення наслідків збоїв

Результати дисертаційного дослідження пройшли всебічну апробацію як у науковому середовищі, так і на експериментальному стенді. Здобуті напрацювання були представлені на низці науково-практичних заходів, зокрема: ХІІ науково-технічній конференції молодих вчених Інституту проблем моделювання в енергетиці ім. Г.Є. Пухова НАН України (Київ, Україна, 2023), круглому столі “Meaningful Artificial Intelligence” (Інститут проблем моделювання в енергетиці ім. Г.Є. Пухова НАН України, Київ, Україна, 2024), науково-практичній конференції “Резильєнтність динамічних систем” (Київ, Україна, 2024), а також на міжнародній конференції “2024 14th International Conference on Dependable Systems, Services and Technologies (DESSERT)” (Athens, Greece: IEEE, Oct. 2024).

Ключові слова: розподілені системи, API, сумісність, несумісні зміни, еволюція програмного забезпечення, еволюція API, CI/CD.

SUMMARY

Yaroshynskyi M.S. A Method for Detecting Incompatible Service Versions Prior to the Deployment of Distributed Software Systems.

Dissertation for the Doctor of Philosophy degree in specialty 122 "Computer Science". – H.E. Pukhov Institute for Modelling in Energy, National Academy of Sciences of Ukraine, Kyiv, 2025.

The dissertation develops a method for detecting incompatible service versions prior to the deployment of distributed software systems. The relevance of the problem is substantiated in the context of asynchronous modifications of application programming interfaces (APIs), which may reduce system reliability. The proposed method is based on formalized API compatibility criteria and ensures automated blocking of incompatible releases, thereby preventing the introduction of breaking changes into the production environment.

The purpose of this research is to develop and substantiate a method for detecting incompatible service versions before deploying distributed software systems. This method enables the early detection of conflicts in inter-service interaction caused by asynchronous interface modifications. The proposed approach aims to improve overall system reliability and stability by preventing failures due to API incompatibilities and by minimizing risks during evolutionary development.

Object of research: detection of incompatible service versions prior to the deployment of distributed software systems, with consideration of API evolution processes.

Subject of research: method for detecting incompatible service versions before deployment of distributed software systems.

Scientific novelty of the work, submitted for defense, is characterized by the following points:

- For the first time, a method has been developed for detecting incompatible service versions prior to the deployment of distributed software systems. Unlike existing approaches, it is based on formalized API compatibility criteria, enabling the automated blocking of incompatible releases, reducing the risk of cascading failures caused by breaking API changes, and ensuring the selection of only compatible configurations while preventing incompatible APIs from entering the production environment.
- A software solution has been designed and implemented to validate the proposed method, which enabled its experimental evaluation on an architectural testbed and confirmed its applicability in real-world distributed systems. The implemented tools provide automated compatibility checks for gRPC- and JSON Schema-based services, integrate into CI/CD pipelines, and support traffic switching to the new release. This has substantiated the increase in system reliability and fault tolerance during the updating of mission-critical services.
- Existing strategies for mitigating the impact of breaking changes in distributed systems have been further advanced through their integration with the proposed method, thereby substantiating the combination with a procedural approach. Such integration ensures multi-level stability verification: at the first level, incompatible API versions are automatically blocked before reaching the test environment, while at the second level, full-scale operational scenarios are reproduced in a controlled environment. This allows not only for the verification of technical interface compatibility but also for assessing service behavior under near-production workloads.

The dissertation systematically analyzes current approaches to managing API evolution in distributed systems, including strategies for minimizing the impact of incompatible changes and methods for maintaining service interoperability. The causes of incompatibilities are categorized, their effects on distributed system operability are revealed, and the limitations of existing DevOps and CI/CD practices are identified, particularly the lack of early detection of critical failures.

The concept of breaking API changes is clarified, and a method for detecting incompatible service versions prior to deployment is substantiated. A novel “API Compatibility Registry” is proposed, which integrates with CI/CD pipelines, automatically verifies inter-service dependencies, and generates deployment scenarios exclusively for compatible configurations. This enables a proactive mechanism of compatibility control that eliminates the occurrence of cascading failures in production environments.

In the introduction, the relevance of the research topic is substantiated, the aim is formulated, the objectives are defined, the subject, object, and research methods are identified, and the main provisions of the scientific novelty and the practical significance of the obtained results are presented.

The first chapter examines the nature and causes of API evolution, including the extension of functionality, migration of the technological base, performance optimization, and design improvements. Particular attention is devoted to the concept of “incompatible changes” and the challenges they create in complex distributed systems. Their impact on the operability of client applications and on the entire service ecosystem is analyzed. A formal model of cascading failures in distributed software systems triggered by incompatible changes is introduced. The chapter also systematizes strategies for risk mitigation: staging testing, automated generation of protocol schemas, the use of adaptive compatibility formats, Blue/Green deployment strategies, and embedding API versions into protocols.

The second chapter presents a systematic analysis of distributed system architectures, which confirmed the diversity of approaches to organizing service interaction. This provided a conceptual foundation for constructing a model of the environment in which API incompatibility risks arise. It is established that under modern deployment processes across different environments, the integration of the method for detecting incompatible service versions prior to the deployment of distributed systems makes it possible to automatically identify conflicts between service versions, thereby preventing the formation of invalid configurations in the production environment. Within the scope of the research, different environments for implementing the method are considered, and the choice of Kubernetes is substantiated as the optimal solution. Its orchestration capabilities provide scalability, lifecycle management of services, and flexible update control, which allow the proposed method to be effectively integrated into deployment processes. Thus, this chapter establishes the scientific and methodological basis for the further software implementation of the developed method and for the experimental verification of its effectiveness.

The third chapter implements an architectural and software approach to constructing an experimental testbed that reproduces the conditions of real-world functioning of distributed systems. The developed method includes a specialized application that automates the process of detecting incompatible API versions even before deployment. This made it possible to verify not only compatibility at the level of individual services but also the correctness of the functioning of the entire distributed software system under cascading dependencies. The constructed testbed architecture demonstrated its suitability both for simulating operational scenarios and for reproducing controlled failures, which is a prerequisite for verifying the proposed method. Existing strategies for minimizing the impact of incompatible API

changes in distributed systems were further developed through their combination with the proposed method, in particular, its integration with the procedural approach.

The fourth chapter describes the experimental study, which confirmed the effectiveness of the proposed method for detecting incompatible service versions prior to deployment. Tool support for compatibility verification of gRPC- and JSON Schema-based services was implemented. During the controlled introduction of incompatible changes, it was shown that the system is capable of timely blocking hazardous releases and preventing cascading failures. The application of traffic-shifting strategies made it possible to assess the robustness of the method, while coordinated service update scenarios confirmed its suitability for systems with a high frequency of change. The experimental results demonstrated that the integration of the proposed approach into CI/CD pipelines increases the reliability and stability of distributed software systems, reducing the risks of failures during API evolution that lead to incompatible changes in inter-service communication.

The conclusions summarize the scientific and practical results obtained in the research.

Relevance of the research. Modern software systems that underpin mission-critical applications and commercial products are increasingly being implemented in the form of distributed architectures. Such approaches provide scalability, fault tolerance, and flexibility of development, but at the same time create new challenges associated with the coordination of numerous components. The problem of minimizing the impact of incompatible API changes in distributed systems becomes particularly acute in critical infrastructure management systems, where the continuity of operation is a decisive condition for reliability..

The evolution of software systems is inevitably accompanied by modifications to application programming interfaces (APIs). Under conditions of asynchronous service updates, this leads to the risk of version incompatibilities,

when the expectations of one component no longer correspond to the data formats or communication protocols of another. Such incompatible changes may trigger cascading failures, which make the stable operation of the entire system impossible.

Therefore, the relevance of this research is determined by the need to create formalized methods and mechanisms capable of ensuring API compatibility control prior to deployment. This makes it possible to improve the reliability and predictability of software system evolution, particularly in domains where the failure of even a single component can result in significant economic losses or security threats.

The practical significance of the dissertation lies in the development of a method that enables engineers and development teams to ensure an enhanced level of reliability and stability in distributed software systems. Applying the method during the pre-deployment phase allows for the automatic detection and elimination of service incompatibilities before they can cause operational failures. This directly reduces the number of incidents associated with inter-service conflicts and shortens the time required for their diagnosis and remediation.

The proposed method is integrated into modern DevOps practices, particularly continuous integration and continuous delivery (CI/CD) pipelines. It provides an additional layer of automated quality control and supports the uninterrupted functioning of services, which is especially important for mission-critical applications. The use of traffic-shifting strategies in conjunction with API compatibility verification minimizes the risks of cascading failures during updates, thereby increasing the reliability of system integration.

The research results can be implemented in both commercial and governmental organizations that operate large-scale distributed software systems. This applies especially to sectors where stability and predictability of operation are critical conditions: energy, transportation, telecommunications, aviation, and

financial systems. Thus, the practical value lies not only in enhancing fault tolerance but also in ensuring economic benefits by reducing the costs associated with failure remediation.

The results of the dissertation research underwent comprehensive approbation both within the scientific community and on the experimental testbed. The obtained findings were presented at a number of scientific and practical events, in particular: the XLI Scientific and Technical Conference of Young Scientists of the H.E. Pukhov Institute for Modelling in Energy of the National Academy of Sciences of Ukraine (Kyiv, Ukraine, 2023); the round table “*Meaningful Artificial Intelligence*” (H.E. Pukhov Institute for Modelling in Energy of the National Academy of Sciences of Ukraine, Kyiv, Ukraine, 2024); the scientific and practical conference “*Resilience of Dynamic Systems*” (Kyiv, Ukraine, 2024); as well as the international conference “*2024 14th International Conference on Dependable Systems, Services and Technologies (DESSERT)*” (Athens, Greece: IEEE, Oct. 2024).

Keywords: distributed systems, API, compatibility, incompatible changes, software evolution, API evolution, CI/CD.

СПИСОК ПУБЛІКАЦІЙ ЗДОБУВАЧА

1. **М.С. Ярошинський**, О.В. Сіроткін, Д.П. Сінько, С.Б. Гунько, Д.О. Манолук, ‘Коректність пласкої класифікації’, Електронне моделювання Т. 45, № 2 (2023) с. 34-43 doi: [/10.15407/emodel.45.02.034](https://doi.org/10.15407/emodel.45.02.034). Фахове видання категорії Б. (Особистий внесок – Брав участь у аналізі матеріалів, розробці та формальному обґрунтуванні математичного апарату для оцінки коректності пласких класифікацій. На основі введених операцій та поняття відносної відстані між класами, запропонував формальну міру відмінності між двома класами).
2. А.М.Примушко, І.В. Пучко, **М.С. Ярошинський**, Д.П. Сінько, ‘Програмний дизайн розподіленої високонавантаженої системи електроенергетичної мережі на базі моделі акторів із застосуванням смарт-контрактів’, Електронне моделювання Т. 46, № 3 (2024) с. 57-72 doi: [/10.15407/emodel.46.03](https://doi.org/10.15407/emodel.46.03). Фахове видання категорії Б. (Особистий внесок – технічне обґрунтуванні архітектурних рішень, що реалізують запропоновану високорівневу концепцію. Визначенні структури даних, що передаються між вузлами системи, запропонував та формалізував JSON-формат із обов'язковим набором параметрів, запропонував модель поведінки IoT-пристроїв у системі, визначивши три ключові стани: Активний, Пасивний та Неактивний).
3. A. Prymushko, I. Puchko, **M. Yaroshynskyi**, D. Sinko, H. Kravtsov, and V. Artemchuk, ‘Efficient State Synchronization in Distributed Electrical Grid Systems Using Conflict-Free Replicated Data Types’, IoT, vol. 6, no. 1, p. 6, Jan. 2025, doi: [10.3390/iot6010006](https://doi.org/10.3390/iot6010006). **Indexed in Scopus Q1**. (Особистий внесок – брав участь у плануванні і проведенні експериментів, брав участь у візуалізації даних, брав участь у

формалізації структури стану вузла та потоків комунікації всередині мережі, формуванні і моделюванні графу роботи мережі).

4. O. Sirotkin, A. Prymushko, I. Puchko, H. Kravtsov, **M. Yaroshynskyi**, and V. Artemchuk, ‘Parallel Simulation Using Reactive Streams: Graph-Based Approach for Dynamic Modeling and Optimization’, *Computation*, vol. 13, no. 5, p. 103, Apr. 2025, doi: 10.3390/computation13050103. **Indexed in Scopus Q2.** (Особистий внесок – брав участь у аналізі та інтерпретації результатів, в розробці математичної моделі для динамічного моделювання).
5. **M. Yaroshynskyi**, A. Prymushko, I. Puchko, O. Sirotkin, and D. Sinko, ‘Akka as a tool for modelling and managing a smart grid system’, *J. Edge Comp.*, vol. 4, no. 1, pp. 105–115, May 2025, doi: 10.55056/jec.822. **Indexed in Scopus.** (Особистий внесок – брав участь у аналізі ключових викликів в управлінні інтелектуальними мережами та в розробці моделі управління інтелектуальною мережею на основі ієрархічної структури акторів Akka, моделював архітектуру мережі — від регіонів до окремих пристроїв, брав участь у концептуалізації та впровадженні підходу, що поєднує моделювання та реальне управління в єдиній акторній архітектурі).
6. **M. Yaroshynskyi**, I. Puchko, A. Prymushko, H. Kravtsov, and V. Artemchuk, ‘Investigating the Evolution of Resilient Microservice Architectures: A Compatibility-Driven Version Orchestration Approach’, *Digital*, vol. 5, no. 3, p. 27, July 2025, doi: 10.3390/digital5030027. **Indexed in Scopus Q2.** (Особистий внесок – брав участь в аналізі існуючих стратегій управління еволюцією API, запропонував підхід з Compatibility-Driven Version Orchestrator, створення математичної моделі поширення збоїв через несумісні зміни API, брав участь в

організації експериментів, проведення експериментів, візуалізація результатів).

7. О.В. Сіроткін, **М.С. Ярошинський**, Д.П. Сінько, С.Б. Гунько, Д.О. Манолук, ‘Моделювання у фазовому просторі під-станів’, Електронне моделювання Т. 47, № 3 (2025), с. 28-45 doi: /10.15407/emodel.47.03.028 Фахове видання категорії Б. (Особистий внесок – брав участь у побудові концептуальної моделі, симуляції моделі)
8. **Ярошинський М.С.**, Пучко І.В. Способи розв’язання проблеми асинхронності зміни прикладного програмного інтерфейса в мікросервісній архітектурі. Електронне моделювання Т. 47, № 4 (2025) с. 57-72 doi: /10.15407/emodel.47.04.057. Фахове видання категорії Б. (Особистий внесок – брав участь в аналізі існуючих стратегій управління еволюцією API, брав участь у створення математичної моделі поширення збоїв через несумісні зміни API).
9. **М.С. Ярошинський**, ‘Захист від атак підробки та перехоплення за допомогою jwt для неконфіденційної інформації’, Матеріали науково-практичної конференції ‘ХЛІ науково-технічна конференція молодих вчених та спеціалістів інституту проблем моделювання в енергетиці ім. Г.Є. Пухова НАН України’, с. 141-145, Україна, 2023, URL: <https://ipme.kiev.ua/konferencii/konferenciya-molodix-vchenix-2023>
10. **Ярошинський М.С.**, ‘Використання генеративного штучного інтелекту для синхронізації задач та документації на проектах, які використовують гнучку підхід розробки ПЗ’, Матеріали круглого столу ‘Meaningful Artificial Intelligence’ Інституту проблем моделювання в енергетиці ім. Г.Є. Пухова НАН України, м. Київ, Україна. с 9 – 12, 2024
11. І.В. Пучко, А.М. Примушко, **М.С. Ярошинський**, Г.О. Кравцов, ‘Підвищення резильєнтності динамічних систем при синхронізації

станів за допомогою CRDT’, Матеріали науково-практичної конференції ‘Резильєнтність динамічних систем, с. 50–52, Київ, Україна, 2024 URL: <https://ipme.kiev.ua/konferencii/naukovo-praktichna-konferenciya-rds-2024>

12. Prymushko A., Yaroshynskyi M., Puchko I. Representation and synchronization of states of distributed electrical grid systems based on conflict free replicated data types. 2024 14th International Conference on Dependable Systems, Services and Technologies (DESSERT), Athens, Greece, 2024, pp. 1-5, doi: [/10.1109/DESSERT65323.2024.11122143](https://doi.org/10.1109/DESSERT65323.2024.11122143).
Indexed in Scopus.

ЗМІСТ

АНОТАЦІЯ.....	2
SUMMARY	9
СПИСОК ПУБЛІКАЦІЙ ЗДОБУВАЧА.....	16
ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ.....	23
ВСТУП	24
Розділ 1. Теоретичні основи управління еволюцією програмних інтерфейсів.....	30
1.1 Технологічні особливості розподілених програмних систем	30
1.2 Причини еволюції API	31
1.3. Поняття несумісних змін	32
1.4. Виклики еволюції АПІ у розподілених системах	33
1.5 Стратегії мінімізації впливу несумісних змін API	36
1.5.1 Процесуальний підхід (staging testing).....	37
1.5.2 Генерація схеми опису протоколу.....	38
1.5.3 Підхід адаптивної сумісності АПІ.....	39
1.5.4 Підхід Blue/Green deployment	41
1.5.5 Включення версії АПІ в протокол для підтримки сумісності..	42
1.6 Аналіз підходів	43
Висновки до розділу	46
Розділ 2. Модель каскадних відмов та метод виявлення несумісних версій сервісів перед розгортанням розподілених систем	48
2.1. Архітектури розподілених систем	48

2.1.1 Архітектура, керована подіями (Event-Driven Architecture).....	50
2.1.2 Стиль просторової архітектури (Space-based architecture)	52
2.1.3 Керована оркестратором сервіс-орієнтована архітектура (Orchestration-Driven Service-Oriented Architecture).....	53
2.1.4 Архітектура на основі сервісів (Service-oriented architecture)..	56
2.1.5 Мікросервісна архітектура (Microservices Architecture).....	59
2.2 Концептуальні основи сумісності в L7-комунікаціях.....	62
2.3 Сучасний стан управління розгортанням розподіленими аплікаціями	67
2.3.1 Контейнеризація	67
2.3.2 Docker	68
2.3.3 Docker Swarm.....	69
2.3.4 Kubernetes.....	70
2.3.5 Apache Mesos, Nomad	72
2.3.6 Ansible	74
2.4 Сервісна сітка як інфраструктурний шар сумісності.....	76
2.5 GitOps як стандарт експлуатації керованої еволюції.....	78
Висновки розділу	80
Розділ 3. Програмна реалізація та архітектура експериментального стенду	82
3.1 Опис методу	82
3.2 Поєднання з процесуальним підходом.....	90
Висновки до розділу	91

Розділ 4. Експериментальне дослідження та верифікація ефективності методу	93
4.1 Постановка експерименту та методологія	93
4.2 Контрольоване внесення несумісної зміни.....	95
4.3 Blue/Green-розгортання за наявності несумісної схеми	96
4.4 Скоординоване розгортання оновлених сумісних версій замість існуючих.....	98
4.5 Метод виявлення несумісних версій сервісів перед розгортанням розподілених програмних систем	100
Висновки до розділу	101
Висновки	103
Список використаних джерел	106
Додаток А.....	122
Додаток Б	126
Додаток В	130
Додаток Г	131

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API – прикладний програмний інтерфейс

CI – Безперервна інтеграція (Continuous Integration)

CD – Безперервна доставка (Continuous Delivery)

SLA – Договір про рівень сервісу (Service Level Agreement)

SLO – Цільовий рівень обслуговування (Service Level Objective)

ВСТУП

Актуальність теми. Сучасні програмні засоби, зокрема ті, що є основою для критично важливих систем та комерційних продуктів, дедалі частіше будуються на основі розподілених архітектур. Така парадигма розробки, як мікросервісна або сервіс-орієнтована архітектура, забезпечує високу масштабованість, відмовостійкість і гнучкість у розробці. Вона дає змогу автономно розробляти, розгортати та оновлювати окремі компоненти (сервіси) системи. Водночас, ця автономність приносить із собою нові виклики, які потребують системного вирішення для забезпечення стабільної та надійної роботи. Особливо гостро ця проблема постає в системах управління, де безперервність функціонування є критично важливою.

Еволюція програмних систем передбачає розвиток їхніх компонентів та прикладних програмних інтерфейсів (API). У розподілених архітектурах оновлення сервісів може відбуватися в різний час, змінюючи їхні функціональні можливості та способи обміну даними. Цей асинхронний процес змін створює ризик виникнення несумісних змін в API, коли один сервіс очікує дані в одному форматі, а інший, оновлений, передає їх уже в новому, непідтримуваному форматі. Такі розбіжності можуть залишитися непоміченими на етапі розробки окремих частин системи.

Ігнорування цього ризику може призвести до серйозних наслідків на етапі розгортання та експлуатації. Виявлення несумісних версій API вже в розгорнутій системі може спричинити часткову або повну втрату працездатності, розрив функціональних ланцюгів. Сучасні практики розробки та розгортання, націлені на автоматизацію, але часто не містять ефективних механізмів для раннього виявлення міжсервісних несумісностей. Це створює "сліпу зону" в процесі забезпечення якості програмного забезпечення, що

вимагає розробки нових методів. Також сучасні методи вимагають значної ручної роботи з виявлення таких несумісностей.

Таким чином, **актуальним науковим завданням** є розроблення методу виявлення несумісних версій сервісів перед розгортанням розподілених програмних систем.

Мета і завдання дослідження. Забезпечення раннього виявлення конфліктів у міжсервісній взаємодії, що виникають внаслідок асинхронних змін в їхніх інтерфейсах, шляхом розробки методу виявлення несумісних версій сервісів перед розгортанням розподілених програмних систем.

Для досягнення поставленої мети сформовано такі взаємопов'язані часткові завдання:

1. Дослідити технологічні особливості еволюції API у розподілених програмних системах.
2. Провести аналіз стратегій мінімізації впливу несумісних змін API у розподілених системах.
3. Розробити метод виявлення несумісних версій сервісів перед розгортанням розподілених програмних систем.
4. Розробити програмне забезпечення для перевірки ефективності методу виявлення несумісних версій сервісів перед розгортанням розподілених програмних систем.
5. Розробити архітектурно-програмний стенд для експериментальної перевірки методу виявлення несумісних версій сервісів перед розгортанням розподілених програмних систем.
6. Обґрунтувати інтеграцію методу виявлення несумісних версій сервісів перед розгортанням розподілених програмних систем з існуючими підходами.

7. Провести експериментальні дослідження із каскадом залежностей, відтворивши сценарії сумісних/несумісних змін.

Об’єктом дослідження є автоматизація виявлення несумісних версій сервісів перед розгортанням розподілених програмних систем з урахуванням процесів еволюції API.

Предмет дослідження – метод виявлення несумісних версій сервісів перед розгортанням розподілених програмних систем.

Методи дослідження – у дисертаційній роботі при розв’язанні поставлених наукових задач комплексно використовувалися методи системного і функціонального аналізу, математичного моделювання, об’єктно-орієнтованого та функціонального програмування, планування наукового експерименту та обробки його результатів.

Наукова новизна здобутих результатів:

1. Вперше розроблено метод виявлення несумісних версій сервісів перед розгортанням розподілених програмних систем, який, на відміну від існуючих, базується на формалізованих критеріях сумісності API, що забезпечує автоматизацію блокування несумісних релізів, знижує ризики каскадних відмов через несумісні зміни API і гарантує вибір сумісних конфігурацій не допускаючи появи несумісних API в робочому середовищі.
2. Розроблено програмне забезпечення для перевірки ефективності методу, що дозволило здійснити його експериментальну апробацію на архітектурно-програмному стенді та підтвердити придатність до практичного застосування у реальних сценаріях розподілених систем. Реалізовані інструменти забезпечують автоматизовану перевірку сумісності gRPC- та JSON Schema-сервісів та перемикання трафіку на новий реліз інтегруються у конвеєри CI/CD, що дозволило обґрунтувати

підвищення надійності і відмовостійкості при оновленні критично важливих інформаційних сервісів.

3. Дістали подальшого розвитку існуючі стратегії мінімізації впливу несумісних змін API у розподілених системах за рахунок їх комбінації із запропонованим методом, а саме поєднання описаного методу з процесуальним підходом. Така інтеграція забезпечує багаторівневу перевірку стабільності: на першому рівні відбувається автоматизоване блокування несумісних версій API ще до моменту їхнього потрапляння у тестове середовище, а на другому — здійснюється відтворення повноцінних сценаріїв роботи системи в умовах тестового середовища. Це дозволяє не лише гарантувати технічну сумісність інтерфейсів, але й оцінити поведінку сервісів під очікуваними навантаженнями.

Особистий внесок здобувача. Наукові результати дослідження, які виносяться на захист, одержані автором самостійно.

Зв'язок роботи з науковими програмами, планами, темами.

Тематика дисертаційної роботи відповідає пріоритетним напрямкам розвитку науки і техніки в Україні. Робота виконувалась відповідно до плану наукових досліджень Інституту проблем моделювання в енергетиці ім. Г.Є. Пухова НАН України, зокрема в рамках НДР «Розвиток наукових засад алгебраїчної теорії сильного штучного інтелекту стосовно кібернетичної безпеки об'єктів критичної інфраструктури в галузі енергетики» (№ ДР 0123U100913, 2023-2027 рр.) та «Розвиток розподіленої енергетики в умовах ринку електричної енергії України з використанням технологій та систем цифровізації. Розділ 1. Організаційні та математичні моделі взаємодії учасників децентралізованого ринку електроенергії» (№ ДР 0125U000237, 2025-2026 рр.).

Практичне значення одержаних результатів.

Розроблений метод дозволяє інженерам і командам розробки підвищити надійність і стабільність програмних систем. Застосування цього методу на етапі підготовки до розгортання дасть змогу автоматично виявляти та усувати несумісності API між сервісами ще до того, як вони можуть спричинити збій у роботі і, як наслідок, зменшити кількість інцидентів, пов'язаних із міжсервісними конфліктами, скоротить час на їх діагностику та виправлення.

Крім того, запропонований підхід сприятиме оптимізації процесів розробки та розгортання (DevOps). Інтеграція методу в існуючі конвеєри безперервної інтеграції та доставки (CI/CD) забезпечує додатковий рівень контролю якості, що є особливо важливим для складних розподілених систем. Це дозволяє зменшити витрати, пов'язані з виправленням помилок після розгортання, і забезпечити безперервну роботу критично важливих сервісів.

В рамках роботи розроблено та реалізовано аплікацію, що дозволяє виявити несумісність версій API сервісів перед розгортанням розподілених програмних систем.

Таким чином, результати дослідження можуть бути використані в комерційних та державних організаціях, які розробляють та експлуатують розподілені програмні системи, для підвищення їхньої відмовостійкості та ефективності.

Апробація результатів дослідження. Результати дисертаційного дослідження пройшли всебічну апробацію як у науковому середовищі, так і на експериментальному стенді. Здобуті напрацювання були представлені на низці науково-практичних заходів, зокрема: XLI науково-технічній конференції молодих вчених Інституту проблем моделювання в енергетиці ім. Г.Є. Пухова НАН України (Київ, Україна, 2023), круглому столі “Meaningful Artificial

Intelligence” (Інститут проблем моделювання в енергетиці ім. Г.Є. Пухова НАН України, Київ, Україна, 2024), науково-практичній конференції “Резильєнтність динамічних систем” (Київ, Україна, 2024), а також на міжнародній конференції “2024 14th International Conference on Dependable Systems, Services and Technologies (DESSERT)” (Athens, Greece: IEEE, Oct. 2024).

Публікації. Результати дисертації представлено у 11 публікаціях, зокрема: 8 статей у провідних фахових виданнях (з них 4 у виданнях, що індексуються у scopus), 4 тези доповідей на конференціях (в тому числі 1 міжнародної конференції, що індексується в scopus).

Обсяг та структура роботи. Дисертація складається зі вступу, чотирьох розділів, висновків, списку використаних джерел і додатків. Дисертаційна робота має 18 рисунків, 1 таблиця, 3 додатків. Список використаних джерел містить 113 найменувань. Загальний обсяг роботи складає 132 сторінки, обсяг основного тексту – 104 сторінки.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ УПРАВЛІННЯ ЕВОЛЮЦІЄЮ ПРОГРАМНИХ ІНТЕРФЕЙСІВ

1.1 Технологічні особливості розподілених програмних систем

Сучасні інформаційні системи еволюціонують під тиском вимог до масштабованості, безперервної доставки змін, міжсервісної узгодженості контрактів взаємодії. Це безпосередньо стосується впровадження розподілених програмних систем для управління й моделювання критичної інфраструктури, зокрема електричних мереж [1–4].

Асинхронна природа еволюційних змін програмних інтерфейсів у розподілених системах може призводити до порушення сумісності в роботі розподілених програмних систем, що формує потребу у формалізованому оркеструванні версій і автоматичному контролі сумісності [4,5]. Ефективне управління версіями протоколів API стає необхідною умовою підтримання працездатності та керованості еволюції систем [5–9].

На цьому тлі підходи, які спираються на власні протоколи синхронізації, складні та низькорівневі [10], що підкреслює необхідність використання узгоджених і загально застосовних протоколів взаємодії в розподілених середовищах. Відсутність механізму координації версій сервісів на тлі зростання частоти асинхронних (і подекуди несумісних) змін API може спричиняти каскадні відмови та простої сервісів у критично важливих доменах — енергетиці, фінтеху, авіоніці [4,5].

У цьому контексті, ефективне управління сервісами залежно від версій протоколів API стає важливою умовою для підтримання працездатності та керованості еволюції систем. Це підкреслює необхідність розробки формалізованого методу оркестрування версій та автоматичного контролю сумісності, який буде універсальним і застосовним у розподілених

середовищах. Таким чином, існує об'єктивна потреба в створенні науково-обґрунтованого підходу, що дозволить проактивно виявляти потенційні несумісності до розгортання, забезпечуючи цілісність та надійність функціональних ланцюгів.

1.2 Причини еволюції API

Еволюція API обумовлена потребами в оновленні функціональності, технологічній адаптації та вдосконаленні дизайну. Александр Лерчер у своїй роботі [8] виділив 4 основні причини проведення змін у API. Першою ключовою причиною є додавання нової функціональності, коли розробники розширюють можливості API для підтримки нових вимог користувачів або бізнес-потреб. Це сприяє зростанню корисності API, але водночас може призвести до зміни його поведінки, що впливає на клієнтські додатки. Другою причиною є зміна технологічної бази, яка включає міграцію на нові хмарні платформи або оновлення версій мов програмування. Це необхідно для підтримки актуальності інфраструктури та безпеки, а також для збереження сумісності з сучасними технологіями. Такі зміни часто вимагають значної перебудови API, що може порушувати попередню функціональність.

Третя причина – покращення наявної функціональності, що може включати оптимізацію продуктивності або злиття схожих робочих процесів для спрощення роботи клієнтів. Хоча це додає цінності API, такі зміни часто змушують клієнтів адаптувати свої додатки до нових налаштувань [8,11,12].

Нарешті, оптимізація дизайну API полягає у видаленні застарілих компонентів або реструктуризації, щоб API був більш зручним і сучасним. Ця оптимізація робить API більш легким у використанні та підтримці, але також може вимагати значних змін у клієнтських додатках [12,13]. API необхідно

розробляти та постійно вдосконалювати, як і будь-яку іншу частину програмної системи. Якщо API задовольняє потреби своїх клієнтів, немає причин для його розвитку. Як показують дослідження, веб-API оновлюються протягом тривалого часу після створення [14,15].

Ці фактори, в сукупності, пояснюють мотивацію для змін API, попри їх потенційні ризики для стабільності роботи клієнтів.

1.3. Поняття несумісних змін

Однією з ключових наукових задач у розробці інформаційних систем є проблема несумісних змін (breaking changes), які порушують зворотну сумісність через видалення або модифікацію елементів API [16]. Ця проблема особливо актуальна для довготривалих проєктів, що мають значну кількість зв'язків між компонентами і потребують регулярних оновлень. Видалення методів, зміна їх сигнатури або модифікація поведінки API може призвести до критичних збоїв у підключених модулях, що ускладнює розгортання нових версій системи. Зважаючи на це, дослідники працюють над методами аналізу змін API, автоматизованим тестуванням зворотної сумісності та розробкою стратегій міграції для мінімізації негативного впливу таких змін.

У цьому контексті зміни API класифікуються на несумісні та сумісні. Несумісна зміна – зміна, яка не є зворотно сумісною. Вона призведе до збою програми, створеної зі старою версією компонента, під новішою версією. Сумісна зміна – зміна, яка є зворотно сумісною. Така зміна може бути вдосконаленням, наприклад, додаванням нових модулів для розширення функціональності компонента. Або це може бути оптимізація продуктивності чи видалення помилок [9].

1.4. Виклики еволюції API у розподілених системах

Автори досліджень [8,16,17] здебільшого аналізують різні аспекти несумісних змін API, їхню частоту, характеристики, ефекти ланцюгової реакції та вплив на програмні екосистеми, причому деякі з них також розглядають вплив на ефективність компаній і зручність для споживачів. Основною проблемою, на якій наголошується, є виклики і ризики, пов'язані з несумісними змінами API під час еволюції програмного забезпечення, де можуть виникати непередбачені наслідки від модифікацій інтерфейсів. Наприклад, зміни в API можуть спричиняти проблеми сумісності, що веде до можливих збоїв у додатках клієнтів, які залежать від цих API для виконання важливих функцій. Такі збої можуть створювати ефекти ланцюгової реакції в програмних екосистемах, впливаючи на залежні сервіси і додатки, як це розглянуто в дослідженнях, зосереджених на технічних наслідках і ефектах у екосистемах [16,18]. Крім того, аналіз мотивації для впровадження несумісних змін [11], показує, що рішення щодо еволюції API іноді можуть віддавати перевагу інноваціям над стабільністю, що може спричинити збої на стороні клієнта. Ці збої, своєю чергою, можуть призводити до фінансових і репутаційних наслідків для компаній, хоча більшість досліджень тут зосереджена переважно на технічних, а не на бізнес-аспектах. В той же час у дослідженнях [17,19,20] несумісні зміни API безпосередньо пов'язуються з потенційними фінансовими збитками та репутаційними втратами, особливо коли збої в сервісах впливають на бізнес-операції клієнтів. Таким чином, хоча значна частина літератури зосереджується на механіці й закономірностях змін API, це непрямо підкреслює значний ризик проблем зі стабільністю в додатках клієнтів, проблему, що ускладнюється відсутністю негайної уваги до стратегій пом'якшення наслідків у ширшій екосистемі.

Розглянемо систему, яка підтримує функціональності A , B та C , де функціональність C забезпечується множиною інформаційних сервісів $S = \{S_1, S_2 \dots S_n\}$. Зміни в API одного з цих сервісів можуть вплинути на працездатність функціональності C , а також на інші функціональності, залежні від C .

Залежність функціональності B від C позначимо як $B \xrightarrow{d} C$, де $d \in \{0,1\}$. Значення $d = 1$ означає, що B залежить від C , тоді як $d = 0$ вказує на відсутність такої залежності.

Припустимо, що $S_i \in S$ — сервіс, API якого зазнало несумісних змін - зміни, які порушують зворотну сумісність через видалення або модифікацію елементів API [16]. Це можна формалізувати як:

$$BreakingChange(S_i) = 1$$

Функціональність C стає повністю або частково недієздатною, якщо хоча б один із інформаційних сервісів S_i зазнав несумісних змін:

$$C_i = \begin{cases} 0, \exists S_j \in S: BreakingChange(S_j) = 1 \wedge \omega_{ji} > 0 \\ 1, otherwise \end{cases}$$

Статус функціональності B залежить від статусу C та глибини розповсюдження d_p при введенні радіуса відмови r (Рисунок 1.1):

$$B_{status} = \begin{cases} 0, d_p(B_i C_i) \leq r \\ 1, otherwise \end{cases}$$

Якщо один з інформаційних сервісів змінює свій API без належного повідомлення або впровадження відповідних стратегій зворотної сумісності, це може призвести до некоректної роботи функціональності чи сервісів, залежних від даного API. Однією з головних проблем, що виникають у такому контексті, є можливість неконтрольованої зупинки роботи важливих вузлів додатку через несумісність версій АПІ або структури даних [16,21–23].

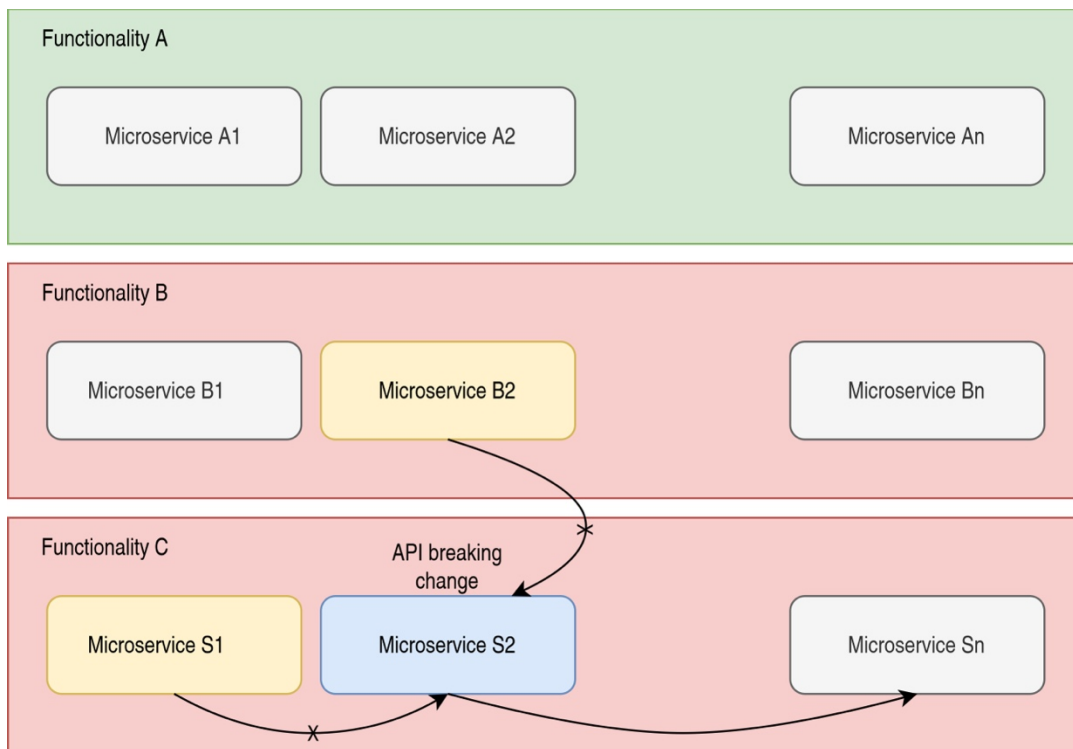


Рисунок 1.1 Схематичне зображення залежностей у багатосервісній системі з несумісними змінами

Наприклад, зміни у форматі відповіді або структурі запитів можуть призвести до неможливості десеріалізації відповіді на стороні споживачів, які очікують дані в іншому форматі, що викликає збої та затримки у загальному процесі обробки запитів [8,16]. Для запобігання подібним ризикам сучасні підходи до еволюційного розвитку API передбачають низку методів, серед яких версіонування API, тестування на основі контрактів, підтримка зворотної сумісності, а також використання спеціальних інструментів для моніторингу стану інформаційних сервісів у реальному часі. Версіонування дозволяє одночасно підтримувати кілька версій API, що забезпечує плавний перехід на нові версії без перерви в обслуговуванні. Тестування на основі контрактів, у свою чергу, дозволяє переконатися, що всі зміни відповідають узгодженим стандартам і не порушують роботу інших компонентів системи. Зворотна

сумісність забезпечує можливість для споживачів використовувати як нову, так і стару версії API, що дозволяє уникнути збоїв.

1.5 Стратегії мінімізації впливу несумісних змін API

Відповідно до законів еволюції програмного забезпечення Лемана [24], промислові програмні системи потребують безперервного супроводу та розвитку для збереження релевантності й відповідності потребам користувачів і ринку. У цьому контексті постачальники API мають своєчасно опрацьовувати критичні зміни, підтримуючи рівновагу між інноваційністю та стабільністю [8]. Зазначений баланс є визначальним для утримання стабільності API з погляду споживачів і запобігання несумісним змінам, здатним порушити роботу клієнтських застосунків.

Еволюційні модифікації API часто спричиняють проблеми сумісності, відмови клієнтських застосунків і пов'язані з ними репутаційні ризики для організацій-розробників. Дефіцит інструментів для прогнозування впливу змін API на клієнтські системи, а також обмежений доступ до даних про реальне використання інтерфейсів додатково ускладнюють підтримання стабільності. Це, у свою чергу, зумовлює потребу у формалізованих методиках запобігання несумісним змінам і в надійних механізмах спостереження та керування їхніми наслідками протягом усього життєвого циклу API.

Сучасні дослідження [8,16,22] сходяться на тому, що успішна еволюція API потребує системних підходів до мінімізації несумісних змін. Далі у роботі узагальнено ключові стратегії, покликані розв'язати цю проблему, зокрема політики версіонування й сумісності, контрактні тести та методи експлуатаційного моніторингу впливу змін.

1.5.1 Процесуальний підхід (staging testing).

Одним із ефективних способів вирішення проблеми неконсистентності API є staging-тестування [25–29], яке проводиться в спеціальному середовищі, що максимально відтворює умови реального виробничого середовища (production). Це середовище, як правило, має ті самі конфігурації, бази даних і сторонні сервіси, що й production, але є ізольованим від користувачів. Staging дозволяє моделювати реальні навантаження та перевіряти сумісність нових змін API з існуючими компонентами системи. Це середовище дає можливість розробникам і тестувальникам (QA) перевіряти зміни як автоматично, так і вручну, що знижує ймовірність непередбачуваних збоїв у клієнтських додатках, оскільки всі зміни тестуються в умовах, наближених до реальних (Рисунок 1.2).

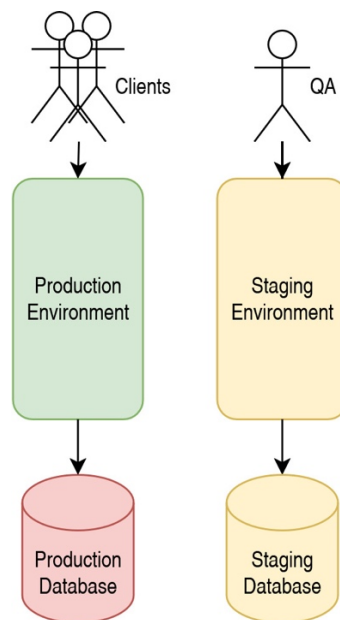


Рисунок 1.2. Схематичне зображення залежностей у багатосервісній системі з несумісними змінами

У staging середовищі перевіряється не тільки функціональність API, але й сумісність його нової версії з існуючими компонентами та залежностями

системи. Це включає тестування взаємодії між сервісами, перевірку коректності форматів даних та функціонування всіх бізнес-процесів, що залежать від API. Автоматизація тестування в staging дає можливість швидко виявляти можливі проблеми з сумісністю і надає розробникам зворотний зв'язок, що є особливо важливим для розподілених архітектур [25–27].

Наприклад, для автоматизації тестування [26,27] можна застосовувати такий інструмент як `rest.io` [30], який підтримує контрактне тестування між сервісами. Це дозволяє перевіряти взаємодію між клієнтами та постачальниками API ще до розгортання змін у робочому -середовищі.

Staging тестування також дає змогу проводити регресійне тестування [28,31], що допомагає переконатися, що зміни не впливають на попередню функціональність. Використання staging середовища для тестування API не лише допомагає уникнути проблем з неконсистентністю, але й сприяє забезпеченню стабільності та надійності системи для кінцевих користувачів [25–28].

1.5.2 Генерація схеми опису протоколу

Один із ефективних підходів до забезпечення сумісності API при постійних змінах — це автоматичне генерування схеми або опису протоколу постачальником API [32–35]. Цей метод передбачає, що постачальник створює актуальну схему, яка визначає структуру та правила взаємодії з API, включаючи доступні ресурси, типи даних та підтримувані версії. Додавання інформації про версію API до згенерованої схеми дає змогу клієнтам чітко розуміти, яку версію вони використовують, і запобігає проблемам із сумісністю, які можуть виникнути через несумісні зміни [34,36].

Усі клієнти API отримують доступ до згенерованих схем, що забезпечує їх узгодженість і зменшує ймовірність помилок, спричинених невідповідністю в структурі даних або методах викликів. Клієнти можуть налаштувати автоматичні перевірки актуальності версії API, звертаючись до центрального репозиторію, де зберігаються останні оновлені схеми. Це дає можливість швидко виявляти зміни, порівнювати поточну версію API з тією, що вони використовують, і вчасно адаптувати свої системи до нових вимог [34,36]. Згідно з проведеним аналізом, автоматизоване оновлення схем із версіонуванням також спрощує процес оновлення клієнтів до нових версій API, зберігаючи сумісність і знижуючи ризики для стабільності клієнтських додатків. Такий підхід дозволяє постачальникам API забезпечити стабільність і зворотну сумісність, мінімізуючи ймовірність збоїв і полегшуючи інтеграцію з новими версіями API.

1.5.3 Підхід адаптивної сумісності API

Використання гнучких форматів даних, які не потребують чітко визначеної схеми, є ефективним способом забезпечення зворотної сумісності API. Формати, такі як JSON або Protocol Buffers (що дозволяють додавати нові поля без порушення сумісності), дають змогу API розширюватися без необхідності кардинальних змін у структурі [7,8,35,37]. Це дає можливість додавати нові елементи даних або функціональність, не порушуючи роботу клієнтів, які використовують старі версії API. Такий підхід робить API більш стійким до змін, що важливо в динамічних системах, де нові вимоги потребують швидкої адаптації.

Перед тим як випустити нову версію API з оновленою структурою даних, проводиться перевірка сумісності нового формату з попереднім. Це

гарантує, що нові формати залишаються зворотно сумісними, і клієнти, які працюють з попередніми версіями API, не стикаються з проблемами через зміни в структурі даних. Наприклад, нові поля в JSON-об'єктах можуть бути опціональними, щоб клієнти могли їх ігнорувати, якщо вони не готові обробляти ці нові поля.

Уникнення несподіваних несумісних змін та забезпечення зворотної сумісності для споживачів API являється основною стратегією, якої дотримуються постачальники API [8]. Це означає, що постачальники прагнуть підтримувати стабільність і мінімізувати ризики, пов'язані з критичними змінами, які можуть вплинути на роботу клієнтських додатків. Згідно з принципом «жодних кардинальних змін без нагальної потреби», розриви сумісності вносяться лише у разі необхідності, що дозволяє уникати непередбачуваних збоїв і зберігати надійність API. Важливо, що постачальники API зазвичай не інформують клієнтів про зміни, які можуть порушити зворотну сумісність, якщо ці зміни не були запитувані самими клієнтами для розширення функціональності. Така стратегія дає змогу постачальникам стабільно підтримувати API, водночас адаптуючи його до нових вимог, коли це справді необхідно.

Застосування цього підходу до форматів, що підтримують зворотну сумісність, значно зменшує ймовірність несумісних змін і дозволяє безболісно переходити на нові версії API. Це також дозволяє постачальникам API оперативно реагувати на нові вимоги, не побоюючись виникнення збоїв у клієнтських додатках. Перевірка сумісності між версіями забезпечує гнучкість і надійність API, що є важливим аспектом для розподілених архітектур, де стабільність і зворотна сумісність є критично важливими.

1.5.4 Підхід Blue/Green deployment

Blue/Green Deployment — це стратегія розгортання програмного забезпечення, яка дозволяє оновлювати додатки з мінімальними ризиками щодо простою і збоїв, забезпечуючи плавний перехід між версіями [38–40]. Вона передбачає використання двох ідентичних середовищ: Green (нове середовище) та Blue (актуальне середовище). Поточне середовище (Blue) обслуговує користувачів, поки нова версія програми розгортається в середовищі Green. Після успішного тестування та перевірки стабільності нової версії трафік може бути поступово або відразу перенаправлений до Green-середовища (Рисунок 1.3).

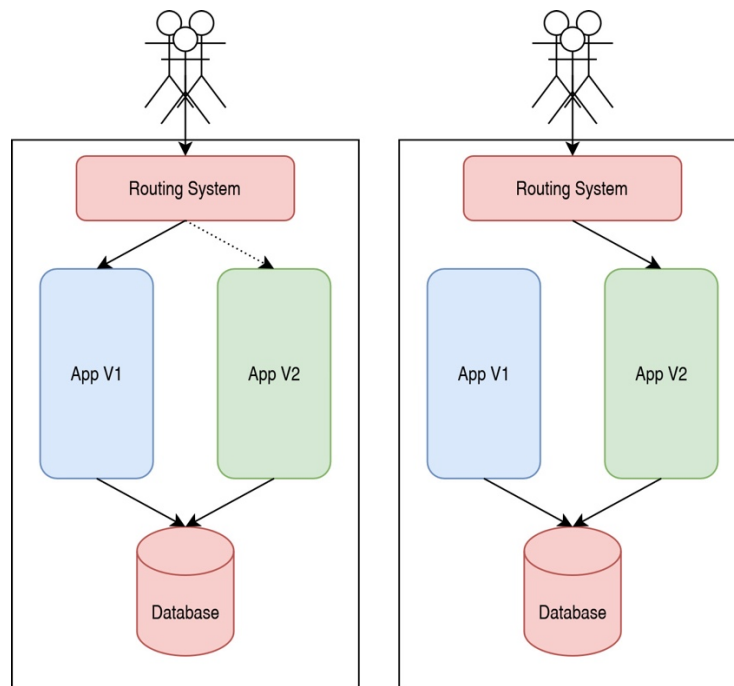


Рисунок 1.3. **Blue/Green deployment** схема

Стратегія Blue/Green Deployment допомагає мінімізувати ризики, пов'язані зі змінами в API, оскільки попередня версія програми залишається доступною та може бути швидко відновлена в разі проблем з новою версією.

Це особливо критично для важливих систем, де збої або довгі простої можуть призвести до значних втрат чи втрати даних. Можливість миттєвого переходу до попередньої стабільної версії забезпечує додаткову надійність і дозволяє командам безпечно впроваджувати нові версії з мінімальним впливом на користувачів [38,40].

1.5.5 Включення версії API в протокол для підтримки сумісності

Включення інформації про версію безпосередньо в API є ефективним способом забезпечення сумісності та управління оновленнями. Це може бути реалізовано шляхом додавання версійних даних у HTTP-заголовки або в саму структуру API, наприклад, у URL чи в тіло відповіді. Таким чином, кожна версія API отримує унікальний ідентифікатор, що дозволяє клієнтам автоматично перевіряти сумісність з поточною версією [41–43].

Клієнтські додатки, отримуючи інформацію про версію, можуть автоматично перевіряти відповідність свого інтерфейсу поточній версії API. Якщо версія API змінена, а клієнтська система не підтримує нову версію, користувачі можуть отримати попередження про несумісність і вжити відповідних заходів, таких як адаптація до нової версії або оновлення залежностей. Це дозволяє запобігти несподіваним збоїв у роботі клієнтських додатків та забезпечує стабільну взаємодію навіть при частих оновленнях API.

Однак підтримка кількох версій API також має свої недоліки, оскільки це збільшує витрати на підтримку старих версій і сповільнює процес впровадження інновацій [8].

1.6 Аналіз підходів

Матеріали [7,8,25–29,32–35,37,44], дозволяють здійснити порівняння підходів, описаних вище (Таблиця 1). Головним спільним недоліком цих підходів є те, що вони не забезпечують імперативного (обов'язкового) виявлення несумісних змін API, унаслідок чого зростає ризик несвоєчасного реагування на порушення сумісності.

Таблиця 1.1 Порівняння методів управління API

Підхід	Переваги	Недоліки
Процесуальний підхід (staging testing)	Дає змогу виявляти та вирішувати проблеми сумісності на етапі тестування. Забезпечує попереднє тестування змін в умовах, максимально наближених до продуктивного середовища	Необхідність залучення додаткових ресурсів і часу для налаштування staging середовища; Необхідність проведення Staging тестування
Генерація схеми опису протоколу	Централізований контроль над версіями API з актуальною схемою; зменшує ризик несумісностей і забезпечує зручність оновлень	Вимагає від клієнтів регулярно оновлювати свої залежності відповідно до змін у схемі

Продовження таблиці 2.1 Порівняння методів управління API

Підхід	Переваги	Недоліки
Підхід адаптивної сумісності АПІ	Дозволяє API еволюціонувати без розривів у сумісності; гнучкість форматів дає можливість додавати нові поля чи зміни з мінімальним впливом на клієнтів	Накопичення технічного боргу через необхідність підтримувати зворотну сумісність; формати, що не мають чіткої схеми, можуть бути менш ефективними для складних даних
Підхід Blue/Green deployment	Забезпечує плавний перехід між версіями з мінімальним ризиком простоїв; можливість миттєвого відкату на попередню стабільну версію сервісу	Необхідність додаткових ресурсів на підтримку двох середовищ; збільшена витрата на інфраструктуру

Продовження таблиці 1.1 Порівняння методів управління API

Підхід	Переваги	Недоліки
Включення версії API в протокол для підтримки сумісності	Надання клієнтам можливість легко визначати сумісність своїх додатків із поточною версією API; забезпечує чіткий контроль версій та швидкий зворотний зв'язок	Необхідність управління кількома версіями API, що може ускладнити обслуговування та потребувати додаткових ресурсів; клієнтам також може знадобитися час на адаптацію до змін; клієнти дізнаються про зміни постфактум; необхідність підтримувати старі версії API невизначений час

Порівняльний аналіз демонструє, що жоден із запропонованих підходів не забезпечує проактивного виявлення несумісностей API на етапі, що передуює розгортанню та функціональному тестуванню. Ці підходи, хоч і важливі, є, по суті, реактивними або вимагають значних інфраструктурних витрат для мінімізації наслідків. Кожна зі стратегій має специфічні переваги та обмеження, які залежать від рівня критичності системи, частоти оновлень та доступності технічних ресурсів. Наприклад, використання staging testing дозволяє виявити проблеми сумісності до виходу змін у виробниче

середовище, але потребує додаткових ресурсів на підтримку даного окремого середовища. Тоді як підхід генерації схеми опису протоколу сприяє підвищенню прозорості взаємодії та передбачуваності змін, однак вимагає дисциплінованого управління версіями. Щодо підходу адаптивної сумісності то слід зазначити, що вона забезпечує гнучкість у внесенні змін без порушення роботи клієнтів, проте може призвести до накопичення технічного боргу. В той же час стратегія Blue/Green deployment мінімізує ризики збоїв під час оновлення, однак потребує подвоєної інфраструктури. Включення ж версії API до протоколу буде забезпечувати явне керування сумісністю, однак ускладнює підтримку попередніх версій.

Висновки до розділу

Обґрунтовано технологічні особливості еволюції API у розподілених архітектурах, яка обумовлена потребами в оновленні функціональності, технологічній адаптації та вдосконаленні дизайну. Визначено, що асинхронна природа цих змін становить ключовий ризик, спричиняючи несумісні зміни (breaking changes), які порушують зворотну сумісність і можуть призводити до неконтрольованої зупинки важливих вузлів.

З'ясовано, що ефект ланцюгової реакції від несумісних змін API може спричиняти каскадні відмови та простої сервісів у критично важливих доменах, таких як енергетика, фінтех та авіоніка. Це підтверджує, що ефективне управління версіями та контроль сумісності є не просто бажаним, а необхідною умовою підтримання працездатності та керованості еволюції розподілених систем.

Проведений порівняльний аналіз існуючих стратегій мінімізації впливу несумісних змін (зокрема, стендового тестування, генерації схем протоколу,

адаптивної сумісності та Blue/Green розгортання) виявив їхні суттєві обмеження. Встановлено, що більшість цих підходів є реактивними, виявляючи несумісність лише на етапі тестування або розгортання, або ж вимагають значних інфраструктурних витрат чи призводять до накопичення технічного боргу.

Обґрунтовано, що існує об'єктивна потреба в створенні науково-обґрунтованого підходу, який дозволить проактивно виявляти потенційні несумісності до розгортання. Це стало основою для формування наукової новизни дисертації — розробки методу, який, на відміну від існуючих, базується на формалізованих критеріях сумісності API і забезпечує автоматизацію блокування несумісних релізів, не допускаючи появи несумісних API в робочому чи тестовому середовищі.

І таким чином, актуальним науковим завданням є розроблення методу виявлення несумісних версій сервісів перед розгортанням розподілених програмних систем.

РОЗДІЛ 2. МОДЕЛЬ КАСКАДНИХ ВІДМОВ ТА МЕТОД ВИЯВЛЕННЯ НЕСУМІСНИХ ВЕРСІЙ СЕРВІСІВ ПЕРЕД РОЗГОРТАННЯМ РОЗПОДІЛЕНИХ СИСТЕМ

2.1. Архітектури розподілених систем

Одним із ключових рішень, яке стоїть перед розробниками програмного забезпечення, є вибір архітектури. Архітектура слугує основою, яка визначає структуру та поведінку інформаційної системи, а також визначає, як окремі компоненти будуть взаємодіяти між собою. Правильно обрана архітектура не лише спрощує процес розробки, але й забезпечує можливість майбутнього розширення та адаптації програмного забезпечення до нових вимог. Від якості прийнятого архітектурного рішення залежить стабільність, масштабованість та зручність використання кінцевого продукту.

Огляд розподілених архітектур у є необхідним для формування моделі середовища, в якому функціонуватиме запропонований метод. Оскільки проблема несумісності API виникає внаслідок міжсервісної комунікації, важливо типологізувати основні архітектурні патерни та формалізувати елементи взаємодії.

У цій роботі будемо виходити з практичного спостереження: сучасні інформаційні інфраструктури — від енергетичних диспетчерських систем до банківських платформ — будуються як розподілені [1,2,4], тобто такі, де окремі компоненти розміщені на різних вузлах мережі та узгоджено виконують спільні задачі, обмінюючись повідомленнями за визначеними контрактами. На прикладному рівні ці компоненти постають як розподілені сервіси: підсистеми з власними станами й даними, які надають функціональність іншим сервісам або кінцевим користувачам через мережеві інтерфейси й еволюціонують у часі, маючи власні життєві цикли версій.

Архітектура програмного забезпечення є основною структурою, яка визначає технічні та операційні вимоги. Вона відповідає за оптимізацію кожного атрибуту додатка, таких як ефективність, керованість, масштабованість, надійність, модифікованість, розгортання та інші аспекти. Саме тому вибір відповідної архітектури є надзвичайно важливим на початковому етапі розробки програмного забезпечення [45].

Існує кілька типів архітектур розподілених систем. Ніл Форд та Марк Річардс виділяють наступні типи архітектури програмного забезпечення [46]: архітектура на основі сервісів, керована подіями архітектура, просторова архітектура, керована оркестратором сервіс-орієнтована архітектура, мікросервісна архітектура.

Лен Басс, Пол Клеменс та Рік Казман [47] також виділяють багаторівневу архітектуру, керовану подіями архітектуру (Publish-Subscribe Pattern), сервіс-орієнтовану архітектуру. Також вони у своїй роботі показують Broker Pattern, Model-View-Controller Pattern, Client-Server Pattern, Peer-to-Peer Pattern, Shared-Data Pattern, Map-Reduce Pattern, Multi-tier Pattern. Які в роботі [46] показані більшими архітектурами, такими як мікросервісна архітектура та архітектура на основі сервісів. Крім того, вони більше схожі на кластерну архітектуру

Еоін Вудс вирізняє наступні види архітектури: монолітна, розподілена, з'єднана через інтернет, інтернет нативна, розумно поєднана [48]. Хоча на думку автора вони й різні, але починаючи від розподіленої фактичної різниці саме в архітектурах не має. Все інше залежить від протоколу комунікації.

2.1.1 Архітектура, керована подіями (Event-Driven Architecture)

Архітектура, керована подіями, становить сучасний підхід до проектування розподілених програмних систем, у якому взаємодія між компонентами організується через обмін дискретними повідомленнями про зміни стану. Подія у цьому контексті — це зафіксований факт, що щось відбулося: створено замовлення, оновлено профіль, завершено розрахунок. Ключова ідея полягає у відокремленні моменту виникнення події від моментів її подальшої обробки: компоненти, які продукують події, не очікують негайної реакції від інших частин системи, а лише надійно фіксують факт змін. Такий підхід підтримує асинхронність, дозволяючи масштабувати оброблення в просторі та часі й знижуючи вимоги до жорсткої координації [46,47,49,50].

Концептуальний фундамент цього стилю визначається трьома ролями: виробник подій, споживач подій і канал поширення подій. Виробник — це компонент, що створює подію як запис про зміну: поряд із назвою події до неї додаються дані корисного навантаження та службові атрибути, зокрема часові мітки, ідентифікатори джерела, а за потреби — версія схеми події. Споживач — це незалежний компонент, що підписується на цікаві йому типи подій і реагує на них відповідно до власної логіки. Каналом називаємо організуючий шар, через який події передаються від виробників до споживачів: він забезпечує буферизацію, впорядкування або принаймні відновлюваність послідовності, а також семантику доставляння, зрозумілу обох сторонам. Завдяки такому поділу обов'язків компоненти не залежать безпосередньо один від одного й можуть розвиватися автономно, зберігаючи спроможність до спільної роботи [46,47,49,50].

Властивості масштабованості та продуктивності в архітектурі, керованій подіями, впливають із самого механізму асинхронного обміну. Оскільки виробник не блокується в очікуванні відповіді, він може обробляти

значно більші обсяги запитів, обмежуючись лише надійною фіксацією подій. Споживачі, у свою чергу, масштабуються незалежно: їх можна додавати або видаляти без впливу на виробників, а навантаження розподіляється між ними природно, згідно з наявними ресурсами. Саме завдяки такому часовому та просторовому роз'єднанню досягається висока пропускна здатність і стійкість до пікових навантажень. Водночас асинхронність вимагає усвідомленого ставлення до узгодженості даних: результати оброблення подій можуть ставати видимими із затримкою, а різні частини системи на коротких інтервалах часу можуть «бачити» дещо відмінні стани. Ця «узгодженість, що встановлюється з часом», є типовою для розподілених систем і потребує чітко визначених інваріантів, які гарантують правильність підсумкових результатів [49–51].

Адаптивність такого стилю проявляється на двох рівнях. На архітектурному рівні система розширюється новими сценаріями: додавання нового споживача, який реагує на наявні події, не потребує змін у виробнику. На організаційному рівні команди можуть працювати автономно, впроваджуючи нові функції через підписку на вже опубліковані події або, за потреби, через введення нових типів подій зі збереженням сумісності з наявними [52].

Архітектурні патерни в межах подійного стилю дають простір для варіацій. Модель «публікація/підписка» формалізує розповсюдження подій до невизначеної кількості зацікавлених споживачів і добре пасує для широкомовних повідомлень про факти. Підхід «джерела подій» (event sourcing) розглядає послідовність подій як первинний записний журнал системи: стан об'єкта можна відтворити, «програвши» всі події, що його стосуються; така інтерпретація створює природну історичність даних і відкриває можливість їхнього аналітичного переосмислення. Часто подійний стиль поєднують з іншими архітектурними підходами: у мікросервісній

архітектурі події виконують роль тканини, що пов'язує автономні служби, тоді як у підході розділення запитів і змін (CQRS) події стають механізмом поширення оновлень до різнорідних проєкцій даних. Таке «вбудовування» не змінює сутності подійної взаємодії, а лише надає їй додаткові засоби для керованого розвитку [46,47,49].

2.1.2 Стил ь просторової архітектури (Space-based architecture).

Більшість бізнес-додатків, заснованих на Інтернеті, дотримуються однакового загального потоку запитів: запит від браузера надходить на вебсервер, потім на сервер додатків і, нарешті, на сервер бази даних. Хоча цей шаблон чудово працює для невеликої групи користувачів, вузькі місця починають з'являтися зі збільшенням навантаження на користувачів, спочатку на рівні вебсервера, потім на рівні сервера додатків і, нарешті, на рівні сервера бази даних. Звичайною реакцією на вузькі місця, пов'язані зі збільшенням навантаження на користувачів, є масштабування вебсерверів. Це відносно легко і недорого, і іноді це допомагає розв'язувати проблеми з вузькими місцями. Однак у більшості випадків високого навантаження на користувачів масштабування рівня вебсервера просто переміщує вузьке місце до сервера додатків. Масштабування серверів додатків може бути складнішим і дорожчим, ніж вебсервери, і зазвичай лише переміщує вузьке місце вниз до сервера бази даних, який ще складніше та дорожче масштабувати. Навіть якщо ви можете масштабувати базу даних, зрештою ви отримаєте топологію у формі трикутника, де найширша частина трикутника – це вебсервери (найлегше масштабувати), а найменша – база даних (найважче масштабувати) [46] (Рисунок 2.1).

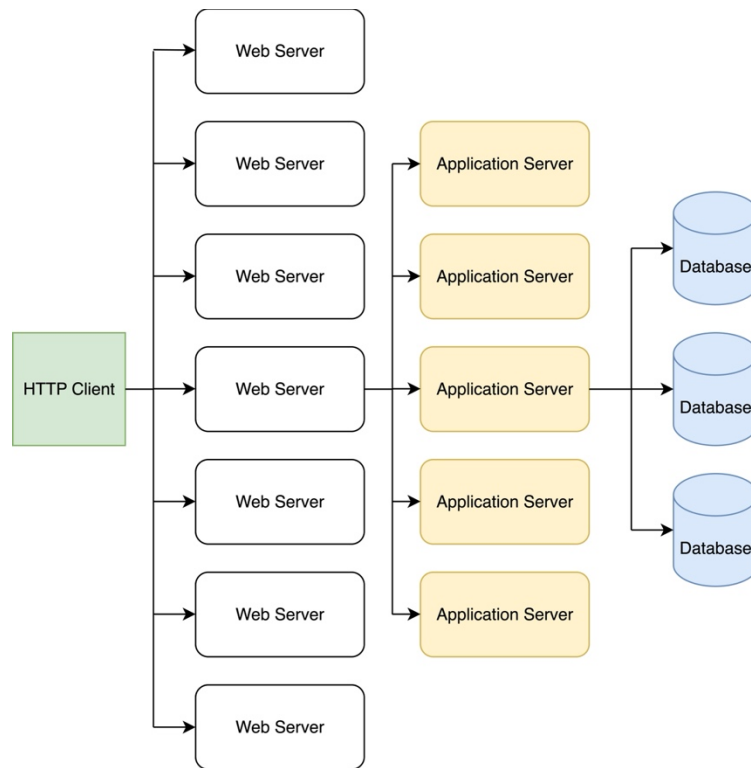


Рисунок 2.1 Просторова архітектура [46]

2.1.3 Керована оркестратором сервіс-орієнтована архітектура (Orchestration-Driven Service-Oriented Architecture).

Керована оркестратором сервіс-орієнтована архітектура репрезентує підхід до проектування розподілених систем, у якому центральний координатор — оркестратор — керує виконанням бізнес-процесів через послідовність взаємодій між окремими службами. На відміну від спонтанної взаємодії компонентів за принципом «кожен із кожним», тут потік виконання є явним, керованим і підзвітним: оркестратор знає, які саме операції мають бути виконані, у якій послідовності, з якими вхідними та вихідними даними і які умови вважаються коректним результатом. Така постановка задачі забезпечує високий рівень контролю за обчислювальним процесом, робить поведінку системи прозорою для аналізу та перевірки, а також створює природні

механізми для підтримання сумісності програмних інтерфейсів у ході еволюції [46,53,54].

У контексті сервіс-орієнтованої архітектури оркестрація означає централізоване узгодження кроків складної операції. Кожна служба зберігає автономію власного життєвого циклу та внутрішньої реалізації, але взаємодіє з іншими за узгодженим сценарієм, який задає оркестратор. Він приймає зовнішній запит, розкладає його на підзадачі, викликає відповідні служби, збирає проміжні результати й ухвалює рішення про подальші кроки або про завершення процесу. Важливо, що оркестратор працює не як «надсервіс», що дублює логіку інших, а як керівник процесу: він не виконує предметні обчислення, а забезпечує правильне зчеплення наявних можливостей системи [46,53,54].

Порівняння з іншими підходами, зокрема з архітектурою, керованою подіями, виявляє різницю в способі координації. Подієвий стиль підкреслює самоорганізацію: служби реагують на факти, що відбулися, не маючи знання про загальний сценарій. Це добре для відкритих екосистем, де невідомо наперед, хто саме зацікавлений у події. Втім, саме ця спонтанність ускладнює гарантії послідовного виконання складних кроків і підвищує вимоги до післярозгортального контролю. Оркестрація натомість робить залежності явними й підконтрольними, що особливо важливо, коли бізнес-процеси потребують жорсткого порядку, а помилка на ранньому кроці дорого коштує. Обидва стилі можуть співіснувати: події інформують про факти, оркестратор — керує причинно-наслідковим ланцюгом там, де це критично [46,53,54].

Стабільність розподіленої системи в керованій оркестратором сервіс-орієнтованій архітектурі підтримується через зменшення невизначеності в поведінці. Коли напрями викликів та умови переходів є відомими, система менш схильна до «випадкових» петлей і неконтрольованих каскадів помилок.

Оркестратор може локалізувати збій, зупинивши прогрес по гілці процесу, не впливаючи на інші, незалежні сценарії. Він також забезпечує впорядкування роботи із зовнішніми ресурсами: узгоджує повторні спроби, обмежує паралелізм там, де існують ризики конфліктів, і керує часовими бюджетами так, аби окремі затримки не накопичувалися до критичних значень [46,53,54].

Практичне застосування цієї парадигми є доцільним у доменах, де процеси мають вагомому нормативну або фінансову ціну помилки: керування замовленнями та розрахунками, ланцюги постачання, надання електронних послуг, агрегування довідкових даних із перевіркою якості. Тут важливі не лише швидкість і масштабованість, а й відтворюваність кроків, можливість прозорого аудиту та чіткі правила відкату в разі виявлення порушень. Оркестратор надає інфраструктуру для таких вимог, перетворюючи процес з неявної взаємодії служб на керований об'єкт із вимірюваними характеристиками [46,53,54] (Рисунок 2.2).

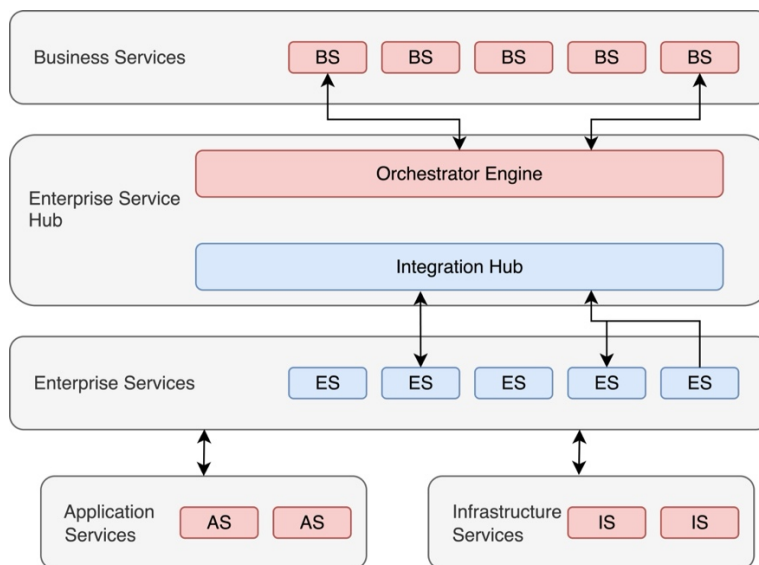


Рисунок 2.2 Керована оркестратором архітектура [46]

2.1.4 Архітектура на основі сервісів (Service-oriented architecture)

Архітектура на основі сервісів репрезентує гібридний підхід до проектування програмних систем, який поєднує модульність і чіткість меж відповідальності з помірним рівнем розподіленості. Йдеться про стиль, у якому система будується як набір самостійних служб (сервісів) із добре визначеними зовнішніми інтерфейсами, проте внутрішня організація кожної служби допускає консолідацію кількох підмодулів, що спільно реалізують цілісний функціональний фрагмент предметної області. Така композиція відрізняється від мікросервісного підходу крупнішим «зерном» декомпозиції: один сервіс охоплює ширший пласт функціоналу, зменшуючи кількість мережових меж і операційних залежностей, але зберігаючи ключові властивості автономії й чіткого контракту взаємодії [46,47].

Співвідношення з мікросервісною архітектурою варто описувати крізь призму балансу витрат і вигод. Мікросервіси максимізують незалежність команд і дрібність оновлень, але ці переваги дістаються ціною зростання кількості мережових викликів, складнішого спостереження та підвищених витрат на узгодження версій інтерфейсів між великою кількістю компонентів. Архітектура на основі сервісів визнає цю «операційну плату» й пропонує компроміс: зменшення числа зовнішніх меж за рахунок укрупнення одиниці розгортання. Вона зберігає ясність контрактів назовні, але допускає більш щільну кооперацію модулів усередині одного сервісу там, де це доцільно з погляду продуктивності, узгодженості даних або команди, що супроводжує систему [47,55,56].

Ключові характеристики цього стилю впливають із вибраного масштабу сервісу. По-перше, архітектурна гнучкість виявляється у здатності адаптувати ступінь розподіленості до реальних потреб. Для сфер, де важливі стабільність і прогнозованість витрат, доцільно обмежити кількість зовнішніх

інтеграційних швів, натомість інвестуючи у чисту внутрішню модульність. Для ділянок, що швидко еволюціонують, межі сервісів можна деталізувати, поступово виділяючи окремі функції в самостійні одиниці. Така «пружність декомпозиції» дає змогу керувати складністю системи у часі: межі не є раз і назавжди заданими, а корегуються за результатами експлуатації [46,47,55,56].

По-друге, прагматичність архітектури на основі сервісів полягає у зниженні загальної вартості володіння. Менша кількість незалежних розгортань означає простішу координацію оновлень, коротший ланцюг пошуку причин збоїв і менше дублікатів поперечних функцій (керування доступом, перетворення форматів, повторні спроби). Водночас система зберігає можливість цілеспрямованого масштабування: якщо певний сервіс стає «вузьким місцем», його можна горизонтально розширити або розділити за доменними підзадачами. Таким чином, порівняно з мікросервісами зменшується «стала частина» витрат (на спостереження, тестування інтеграцій, узгодження багатьох версій), а «змінна частина» залишається керованою завдяки відокремленню [46,47,55,56].

По-третє, рівень розподіленості у цьому стилі є помірним. Зовнішня взаємодія між сервісами спирається на стабільні інтерфейси з чітко визначеною семантикою, тоді як значна частина внутрішньої координації переноситься за межі мережі у внутрішній процес сервісу. Це зменшує експозицію системи до класичних викликів розподілених обчислень — мережевої непевності, накопичення затримок, повторних спроб — і разом із тим не позбавляє можливості будувати цілісні наскрізні сценарії. Інакше кажучи, стиль обирає інформаційне приховування як базовий механізм керування складністю: назовні відкриваються лише ті аспекти поведінки, які справді необхідні для інтеграції [46,47,55,56].

У сервісній архітектурі зовнішній контракт кожного сервісу стає первинним носієм зобов'язань перед іншими частинами системи. З огляду на укрупнене «зерно» декомпозиції, кількість таких контрактів менша, ніж у мікросервісному підході, що полегшує їх системне управління. Це створює сприятливі умови для дисципліни еволюції інтерфейсів: перевага надається розширювальним змінам, що не руйнують чинних взаємодій; для несумісних змін застосовуються контрольовані перехідні періоди з адаптерами, узгодженими термінами міграції та прозорими маркерами версій. Зменшення кількості зовнішніх швів знижує імовірність ланцюгових несумісностей, коли дрібна поправка в одному компоненті спричиняє каскад оновлень в інших.

Звідси випливають і особливості розгортання. Оскільки сервіс об'єднує декілька функціональних підмодулів, оновлення можна планувати більшими, але рідшими кроками, фокусуючись на ретельній перевірці сумісності єдиного зовнішнього контракту. Поступове впровадження змін — спершу для обмеженої частки трафіку чи вибіркових сценаріїв — дає змогу оцінити вплив нової версії на фактичні взаємодії, спираючись на спостережувані показники часу відповіді, частки відмов перевірки формату та стабільності бізнес-правил. У разі виявлення відхилень повернення до попереднього стану не потребує широкої координації з багатьма залежними командами: корекції зосереджені в межах одного сервісу, що скорочує час відновлення та підвищує передбачуваність операцій.

Хоча архітектура на основі сервісів є розподіленою архітектурою, вона не має такого ж рівня складності та вартості, як інші розподілені архітектури, такі як мікросервіси або архітектура, керована подіями, що робить її дуже популярним вибором для багатьох бізнес-додатків [46,47].

2.1.5 Мікросервісна архітектура (Microservices Architecture).

Назва та опис цього архітектурного стилю були визначені у блог-пості Мартіна Фаулера та Джеймса Льюїса. Вони дали наступне визначення: підхід до розробки однієї програми як набору невеликих служб, кожна з яких працює у своєму власному процесі та спілкується за допомогою легких механізмів, часто HTTP ресурсного API. Ці послуги побудовані навколо бізнес-можливостей і можуть незалежно розгортатися за допомогою повністю автоматизованого механізму розгортання. Існує мінімум централізованого керування цими службами, які можуть бути написані на різних мовах програмування та використовувати різні технології зберігання даних [14].

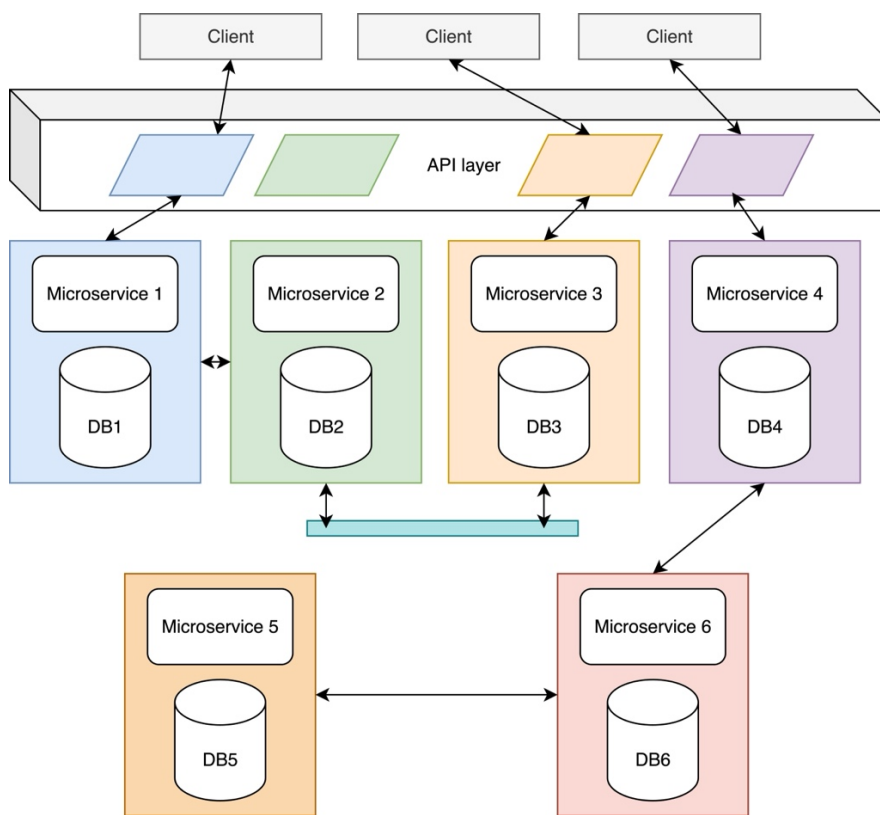


Рисунок 2.3 Мікросервісна архітектура [46]

Мікросервісні архітектури стають популярними альтернативами існуючим парадигмам розробки програмного забезпечення, особливо для розробки складних і розподілених програм. Мікросервіси виникли як шаблон

архітектурного проєктування, спрямований на розв'язання проблем масштабованості та полегшення обслуговування онлайн-сервісів [57].

Мікросервіс — це програмний блок, який можна створювати, ініціалізувати, дублювати та знищувати незалежно від інших мікросервісів тієї ж системи. Крім того, мікросервіси можна розгортати на гетерогенних платформах виконання в мережі. Використання мікросервісів забезпечує високу масштабованість і гнучкість великомасштабних і розподілених програмних систем. [57]

Стиль архітектури мікросервісів зараз є найпоширенішою основою для розробки програмних систем, які є одночасно масштабованими та простими в обслуговуванні. [58]

Розділення програмних систем на слабо зв'язані мікросервіси, слугує покращенню їхньої підтримки. Це дозволяє легше оновлювати та змінювати окремі компоненти системи без впливу на інші частини, що значно спрощує процес обслуговування та розвитку. [8]

Провідні компанії використовують мікросервісну архітектуру для проведення підприємницької діяльності. Наприклад компанія AirBnB перейшла з використання монолітної архітектури до мікросервісної починаючи з 2017 року [59]. Корпорація Uber є також однією з провідних у сфері інтелектуальних технологій із застосування мікросервісної архітектури [60]. Корпорація Netflix також перейшла на мікросервісну архітектуру у 2018 році задля підвищення стабільності та масштабованості платформи [61].

Провідні провайдери інфраструктури для запуску і підтримки програмних продуктів створюють спеціалізовані платформи спеціально для підтримки мікросервісної архітектури. Microservices Engine (MSE) від Alibaba Cloud надає високопродуктивні та високодоступні можливості хмарних служб корпоративного рівня, такі як реєстрація та виявлення служб, керування

конфігурацією, підключення до шлюзу та керування послугами. Центр реєстрації та налаштування повністю керований і сумісний з Nacos, ZooKeeper або Eureka. Шлюзи побудовані на основі Istio та сумісні зі стандартом Kubernetes Ingress. MSE забезпечує покращене керування послугами на основі фреймворків мікросервісів з відкритим кодом, таких як Spring Cloud і Apache Dubbo. MSE допомагає використовувати технології з відкритим вихідним кодом для створення власної системи мікросервісів з легкістю [62]. Або від Google на платформі Python було створено Microservices Architecture on Google App Engine [63].

Існування інструментів інфраструктури cloud native [64]. Наприклад інструмент Jaeger був спеціально створений для того, щоб відстежувати зв'язки між сервісами та метрики в мікросервісній архітектурі Uber [65,66].

З огляду на емпіричні спостереження та аналіз описаних архітектур, переважна більшість сучасних розподілених архітектур реалізує міжкомпонентну взаємодію на рівні прикладного протоколу HTTP або його похідних (HTTP/2, HTTP/3/QUIC; RPC-фреймворки на кшталт gRPC поверх HTTP/2; GraphQL/SSE/WebSocket із початковим HTTP-узгодженням). Надалі буде формалізовно саме такі взаємодії, трактуючи їх як репрезентативний випадок класу L7-комунікацій [67].

Під терміном L7-комунікації розуміють взаємодію сервісів на прикладному рівні (Application Layer, Level 7) мережевої Моделі Взаємодії Відкритих Систем. Це концептуальна модель, що використовується для опису того, як різні мережеві пристрої та програмні додатки взаємодіють між собою [67].

Зміщення фокуса на HTTP-сім'ю не звужує загальності висновків. По-перше, значна частина подієво-орієнтованих та потокових сценаріїв реалізується в межах того ж самого набору комунікаційних протоколів (SSE,

WebSocket, gRPC-streams), а отже підпадає під ті самі вимоги до сумісності, ідемпотентності та керування затримками (включно з особливостями потокового мультиплексування в HTTP/2 та усуненням TCP head-of-line blocking у QUIC). По-друге, навіть коли у внутрішньому контурі застосовуються брокери повідомлень із нефамільними протоколами (AMQP, MQTT, власні двійкові протоколи), на периферії системи переважає HTTP-експозиція.

2.2 Концептуальні основи сумісності в L7-комунікаціях

Попередня частина окреслює актуальність забезпечення сумісності на рівні L7-комунікацій [67] у контексті розгортання інформаційних сервісів у розподілених системах. На відміну від нижчих рівнів мережевої моделі, де сумісність визначається електричними, каналними чи транспортними характеристиками, на прикладному рівні об'єктом узгодження стає «контракт взаємодії» між постачальником і споживачем програмного інтерфейсу. Саме контракт, будучи носієм як структурних, так і поведінкових вимог, фокусується на сумісності API й таким чином визначає можливість безпечної еволюції компонентів, які взаємодіють у неоднорідному середовищі.

Аналіз протоколів верхнього рівня дає змогу уточнити, як саме контракт матеріалізується для різних стилів взаємодії. У випадку HTTP/2 контракт для типового REST-інтерфейсу [23] включає формальний опис структури запитів і відповідей у термінах JSON Schema або еквівалентних специфікацій (наприклад, OpenAPI як носій схем та правил серіалізації) [68,69]. Семантика поведінки фіксує ідемпотентність методів, припустимі стани ресурсу та очікувані коди відповіді, включно з помилковими сценаріями і політикою повторних запитів. На цільовому рівні обслуговування (SLO) та

договору про рівень сервісу (SLA) контракт задає цільові показники доступності, граничну затримку обробки для критичних маршрутів, ліміти на розмір повідомлень і правила керування потоком, що узгоджуються з можливостями мультиплексування HTTP/2. Сумісність тут визначається не лише стабільністю схеми, а й збереженням інваріантів статус-кодування та семантичних обмежень, наприклад, заборонаю зміни ідемпотентної операції на неідемпотентну в межах мінорного оновлення.

Для HTTP/3, який використовує транспорт QUIC, контракт взаємодії на рівні прикладних відомостей зберігає ту ж логічну структуру, однак додає експліцитні припущення щодо характеристик мережевого середовища, зокрема очікування щодо стійкості до втрат пакетів і поведінки потоків при змішаному трафіку. Семантика методів і ресурсів залишається ключовою складовою, але з точки зору SLO контракти для HTTP/3 можуть фіксувати інші профілі затримок і політики повторної передачі, що корелюють із незалежністю потоків у QUIC. Сумісність у такій конфігурації охоплює не лише еволюцію схеми та поведінки, а й стабільність обумовлених параметрів продуктивності, без яких споживач не досягає передбачуваної якості обслуговування [70–72].

У випадку gRPC контракт взаємодії фіксується через Protocol Buffers, де схема описує типи повідомлень, служби та методи, включно з полями, мітками й варіантними структурами. Семантичний компонент контракту уточнює очікування щодо ідемпотентності RPC-викликів, атомарності серверних змін стану, детермінованості порядку обробки та інваріантів сумісності еволюції повідомлень (зокрема, заборони повторного використання вилучених ідентифікаторів полів, допустимості додавання optional-полів та розширень без порушення зворотної сумісності). На рівні SLO контракт для gRPC містить вимоги до граничної латентності для unary-викликів і стійких

бюджетів затримок для потокових сценаріїв, граничні значення пропускну здатності, політику backoff для ретраїв і допустимі рівні помилок транспортного та прикладного рівнів. Визначеність таких параметрів дозволяє формально перевіряти як синтаксичну, так і поведінкову сумісність під час автоматизованого розгортання [32,34].

Подієві потоки як специфічний стиль L7-комунікацій потребують окремого окреслення контракту. Для Server-Sent Events схема даних зазвичай представлена структурованими JSON-повідомленнями з фіксацією типів подій та їх атрибутів; семантика визначає гарантії порядку, політику повторного під'єднання клієнта, відновлення позиції потоку через ідентифікатори подій та вимоги до ідемпотентності обробки на стороні споживача. SLO для SSE артикулює очікувані інтервали «тепла каналу», цільову відсутність розривів сесії і граничну затримку доставлення подій від моменту фіксації на сервері до появи в клієнта. Сумісність у цьому випадку означає незмінність формату подій та стабільність семантичних угод щодо відновлення потоку під час еволюції протоколу застосунку [49].

Для WebSocket контракт взаємодії об'єднує формальний опис кадрів повідомлень (часто у вигляді JSON-схем або бінарних протоколів поверх WS), специфікацію лексики подій і оркестрацію діалогів у вигляді автоматів станів, що визначають дозволені послідовності повідомлень. Семантика встановлює інваріанти порядку, підтвердження доставки на прикладному рівні та політики відновлення після розривів, тоді як SLO акцентує стабільність бітрейту, цільові затримки для інтерактивних сценаріїв та обмеження на розмір і частоту кадрів, що особливо критично для мобільних і високочастотних застосунків. У такому разі сумісність зводиться не лише до стабільності формату, а й до збереження «мови протоколу» застосунку, коли додавання нових типів подій не змінює обов'язкових діалогових шаблонів [73–76].

Стосовно gRPC-streams контракт взаємодії уточнює типи потоків (server-streaming, client-streaming, bidirectional), а також семантику керування потоком і backpressure на прикладному рівні. Схема зберігається у Protocol Buffers із тими ж правилами еволюції, проте для потоків критичним є фіксування інваріантів порядку, маркерів завершення та сигналів помилок, що дозволяє споживачеві розрізняти кінцеві та проміжні стани. На рівні SLO додатково фіксуються гарантовані темпи генерації подій, допуск до «бурстів», час до першого байта та стабільність пропускної здатності впродовж сесії. Збереження цих інваріантів є необхідною умовою сумісності під час розширення чи оптимізації серверної реалізації [32,34,77].

Деталізація поняття контракт взаємодії у загальному вигляді поєднує три компоненти. Схема формує синтаксичний шар, у якому визначаються структури повідомлень, типи, обов'язковість і обмеження, що є базою для статичної перевірки змін і генерації артефактів клієнт-сервер. Семантика становить поведінковий шар, який фіксує ідемпотентність і транзакційність операцій, передумови та постумови викликів, інваріанти порядку та узгодженості, гарантії доставки та правила обробки помилок; саме семантика перетворює синтаксично коректну взаємодію на функціонально правильну.

З позиції узгодження з засадами сумісності програмних інтерфейсів, контракт слугує формальним посередником між еволюцією інтерфейсу та гарантіями якості обслуговування, забезпечуючи розв'язання напруження між необхідністю впровадження нових можливостей і стабільністю існуючих інтеграцій. Принципи чіткої стратифікації змін, розділення синтаксичних і поведінкових інваріантів, а також керованості ризиками за допомогою політик SLO реалізуються в L7-комунікаціях через практики контрактного тестування, сумісних версіонувань та інженерні політики повторних спроб і таймаутів, які повинні бути явно зафіксовані в контракті.

Для ілюстрації зв'язку між компонентами пропонується текстовий опис діаграми рівнів «протокол → контракт → політика». На першому рівні розміщено транспортно-прикладний протокол, що забезпечує базові механізми доставки, мультиплексування і керування потоком. Другий рівень становить власне контракт, який інкапсулює схему повідомлень і семантику поведінки, завдяки чому повідомлення, доставлені протоколом, набувають прикладного значення. Третій рівень представлений політиками, що задають SLO, правила маршрутизації версій, бюджет повторних спроб, дедлайни, а також механізми керування ризиками на кшталт circuit-breaker і rate-limiting. Взаємодія рівнів є двоспрямованою: протокол накладає обмеження на контракт (наприклад, максимальний розмір кадру або модель потоків), тоді як контракт і політика визначають, як саме використовувати можливості протоколу, щоб досягти оголошених гарантій якості та зберегти сумісність під час еволюції. У процесі розгортання інформаційних сервісів це зводиться до перевірки того, що обрана комбінація протоколу, контракту та політики забезпечує виконання критеріїв прийнятності змін, сформульованих як для синтаксичної, так і для поведінкової та експлуатаційної сумісності.

Завершуючи, зазначу, що контракт взаємодії є центральним поняттям методів розгортання з урахуванням сумісності програмних інтерфейсів розподілених систем. Він формалізує очікування споживача і зобов'язання постачальника на трьох взаємопов'язаних шарах — схеми, семантики та політик SLO/SLA — і тим самим надає операціоналізований критерій, за яким еволюцію можна автоматизовано перевіряти і контролювати під час CI/CD-процедур. Для протоколів комунікації така формалізація дозволяє уніфікувати підходи до оцінки впливу змін, забезпечуючи збереження функціональної коректності, передбачуваності продуктивності та узгодженості інтеграцій у гетерогенних середовищах. У рамках загальної методології дисертації це

створює підґрунтя для оркестрації версій і контрольованого впровадження оновлень, де критерії сумісності стають першокласними артефактами процесу розгортання.

2.3 Сучасний стан управління розгортанням розподіленими аплікаціями

Керування розгортанням розподіленими аплікаціями сьогодні формується на перетині трьох взаємопов'язаних напрямів: контейнеризації як стандартизованого способу пакування та ізоляції робочих навантажень, кластерних систем оркестрації та планування ресурсів, що підтримують модель бажаного стану та самовідновлення, і інструментів автоматизації та керування конфігураціями, які забезпечують відтворюваність змін і контроль політик.

2.3.1 Контейнеризація

Контейнеризація — це спосіб запуску кількох програмних додатків на одній машині. Кожна програма працює в ізольованому середовищі, яке називається контейнером. У кожному контейнері є всі файли та бібліотеки, необхідні для нормальної роботи [15,47,48,78]. Технологія простору імен дозволяє безпосередньо розгортати кілька контейнерів на одній машині та обмінюватися ресурсами, а контейнери можна створювати, розгортати та знищувати швидше. Ці переваги роблять контейнер легким, ефективним і недорогим. У міру того, як контейнерна технологія набуває все більшої популярності серед розробників, багато організацій починають використовувати технологію контейнерів для розгортання додатків для роботи. Зі збільшенням кількості контейнерів управління та оркестрування контейнерів стає складним завданням. Щоб працювати та керувати кількома

контейнерами, потрібна платформа оркестрації контейнерів для запуску та спільного керування кількома екземплярами контейнерів. [15]

Для управління контейнерами та платформами існує досить багато інструментів. Для створення, розгортання та управління контейнерами використовують наступні інструменти: Docker, Podman, Cri-O. Системи оркестрації контейнерів, що автоматизує розгортання, масштабування та управління контейнеризованими додатками: Kubernetes, OpenShift, Mesos/Marathon. [15,79–81].

На поточний момент найпопулярнішими контейнерами та платформами для управління контейнерами та контейнеризацією є Docker [14,15,47,57] та Kubernetes [8,13,15,47,58].

2.3.2 Docker

Docker — це відкрита платформа для розробки, доставки та запуску програм. Docker дозволяє відокремити ваші програми від інфраструктури, щоб ви могли швидко доставляти програмне забезпечення. За допомогою Docker ви можете керувати своєю інфраструктурою, так само як і своїми програмами. Використовуючи переваги методології Docker для доставлення, тестування та розгортання коду, ви можете значно скоротити затримку між написанням коду та його запуском у виробництві [82]. Docker працює з програмами для пакування та їх залежностями. Схема контейнеризації, використана Docker, дозволяє створювати кілька контейнерів з використанням одного образу та спільної операційної системи. Сам по собі Docker не може автоматично планувати та масштабувати контейнери, для максимального використання можливостей та переваг контейнерів потрібна система управління контейнерами, і для цього застосовують Kubernetes [15].

2.3.3 Docker Swarm

Docker Swarm (режим swarm у Docker Engine) — це вбудований механізм кластеризації та оркестрації контейнерів, який формує «рой» із вузлів двох ролей: менеджери (керують станом) і воркери (виконують задачі). Глобальний стан кластера зберігається та узгоджується на менеджерах за алгоритмом консенсусу Raft, що забезпечує відмовостійкість керуючої площини та вибір лідера за втрати вузлів. [83]

Базові абстракції Swarm — сервіс і задача (task). Сервіси бувають реплікованими (вказується кількість екземплярів) та глобальними (по одному екземпляру на кожному вузлі). Менеджер сприймає декларативний опис як «бажаний стан» і розкладає задачі по вузлах; призначена задача не мігрує, а або виконується на заданому вузлі, або перезапускається/відновлюється за політикою контролера. [83,84]

Мережна підсистема використовує оверлей-мережі та спеціальну ingress-мережу з routing-mesh на основі IPVS для прийому трафіку на будь-якому вузлі кластера й прозорого проксування до активних екземплярів сервісу. Swarm підтримує два режими ендпойнтів: vip (віртуальна IP-адреса сервісу з внутрішнім балансуванням) і dnsrr (DNS round-robin без VIP), що дає змогу підбирати спосіб маршрутизації залежно від вимог до з'єднань і балансування. За потреби інтегрується зовнішній балансувальник (наприклад, HAProxy), який спрямовує трафік на опубліковані порти будь-яких вузлів рою. [84]

Для керування конфігурацією та секретами передбачені об'єкти configs і secrets: конфіденційні дані доступні лише сервісам у режимі swarm і доставляються контейнерам через захищені канали/файлові дескриптори; неконфіденційні конфіги дозволяють відокремити артефакт образу від параметрів розгортання. [83,84]

Операційні механізми включають поетапні оновлення й відкат: політики update/rollback задають паралелізм оновлень, затримки, реакцію на збої (pause/continue/rollback) та порядок start-first/stop-first, що дає змогу досягати безперервності сервісу під час релізів. Розміщення керується ресурсними лімітами/резерваціями, обмеженнями (constraints) за мітками вузлів і м'якими преференціями розподілу (placement preferences) [83,84] .

2.3.4 Kubernetes.

Kubernetes — це відкрита платформа для управління контейнеризованими робочими навантаженнями та супутніми службами. Основні характеристики цієї платформи включають кросплатформеність, розширюваність, ефективність використання декларативної конфігурації та автоматизації. Ці властивості роблять Kubernetes ключовим елементом у сучасних підходах до управління інфраструктурою, забезпечуючи масштабованість і надійність системи, що підтверджується її стрімким розвитком у межах глобальної екосистеми.

Назва Kubernetes походить від грецького слова, що означає «керманич» або «пілот». Ця платформа була відкрита компанією Google у 2014 році та базується на 15-річному досвіді Google в управлінні масштабними робочими навантаженнями. Kubernetes впроваджує найкращі практики та ідеї, що були розроблені як всередині компанії, так і спільнотою розробників. [85]

Еволюція управління робочими навантаженнями починалася з використання фізичних серверів, де застосунки запускалися без обмежень на використання ресурсів, що призводило до неефективного розподілу та використання цих ресурсів. Віртуалізація стала вирішенням цієї проблеми,

дозволяючи запускати декілька віртуальних машин на одному фізичному сервері, забезпечуючи ізоляцію та безпеку застосунків.

Згодом контейнери стали популярним інструментом для розгортання та управління застосунками, пропонуючи легшу і більш ефективну альтернативу віртуальним машинам. Контейнери дозволяють ізолювати застосунки та їхні залежності, забезпечуючи гнучкість у переміщенні між різними середовищами та постачальниками хмарних послуг.

Kubernetes з'явився як відповідь на потребу в ефективному управлінні контейнерами на масштабованому рівні. Він надає інструменти для автоматизації розгортання, масштабування, оркестрації та самозцілення контейнерів, що робить його важливим компонентом у сучасних архітектурах інформаційних сервісів. Ця платформа підтримує широкий спектр сценаріїв використання, від простих веб-додатків до складних розподілених систем із різними типами навантаження.

Завдяки своїй архітектурі Kubernetes не є жорстко зв'язаною системою, що дозволяє користувачам налаштовувати та інтегрувати різноманітні інструменти та сервіси відповідно до своїх потреб. Це дає можливість створювати індивідуальні рішення, використовуючи блоки, що надає Kubernetes, і забезпечує високий рівень гнучкості та контролю над інфраструктурою.

Для полегшення управління системою kubernetes існує багато інструментів. Portainer [86] - інструмент для управління Docker та Kubernetes через веб-інтерфейс. Rancher [87] - управління Kubernetes, що дозволяє управляти декількома кластерами Kubernetes з одного інтерфейсу. k9s - це термінальний користувацький інтерфейс (Terminal UI) для взаємодії з кластерами Kubernetes [88]; Helm [89] - менеджер пакетів для Kubernetes, що

дозволяє за допомогою шаблонів встановлювати, оновлювати та керувати складними додатками в Kubernetes.

2.3.5 Apache Mesos, Nomad

Apache Mesos — це кластерний менеджер із дворівневим плануванням, спроектований як тонкий шар для «тонкого» (fine-grained) спільного використання ресурсів між різнорідними фреймворками (Spark, Hadoop, Marathon тощо). У базовій архітектурі майстер Mesos керує агентами та взаємодіє з фреймворками, кожен із яких має власний планувальник і виконавець; ресурси розподіляються через механізм «resource offers», коли фреймворки обирають, які запропоновані ресурси використати. Така двоступенева модель дає змогу масштабовано поєднувати різні типи навантажень і політики, не нав'язуючи єдиної стратегії планування на рівні кластера. Класичний опис підходу наведено в роботі Hindman [79,90], а поточні деталі архітектури та компонентів — в офіційній документації проєкту. [90,91]

На практиці поверх Mesos часто використовують фреймворки-оркестратори. Marathon [92]— один із найуживаніших для довготривалих сервісів і контейнерних робочих навантажень; він реалізує високодоступний запуск та відновлення задач, а також інтегрується в екосистему DC/OS [93], побудовану на Mesos. Такий стек дозволяє експлуатувати гібридні сценарії (контейнеризовані та неконтейнеризовані процеси) в єдиному пулі ресурсів [90,94].

Nomad від HashiCorp — це загальнопризначений планувальник/оркестратор із простою клієнт-серверною архітектурою (сервери — контрольна площа, клієнти — виконання), який підтримує різні типи навантажень: контейнеризовані, віртуалізовані та «standalone» (у т. ч.

Windows/IIS, Java, QEMU) [90,95]. Архітектура передбачає регіони та дата-центри, вбудовану федерацію кластерів і реплікацію стану, що полегшує міждата-центрові розгортання. Модель планування включає кілька спеціалізованих планувальників: `service` (для довготривалих сервісів), `batch` (швидке розміщення пакетних задач), `system` та `sysbatch` (системні задачі на всіх вузлах), що дозволяє узгоджувати різномірні профілі роботи в одному кластері. [90,95,96]

Nomad тісно інтегрується з Consul для сервіс-дискавері, перевірок стану та багатодата-центрових топологій; це забезпечує автоматичну реєстрацію сервісів і надає DNS-інтерфейс, а також підтримує сценарії канарних/blue-green оновлень на рівні специфікацій `job`. У продакшн-довідниках HashiCorp наведено референс-архітектури для високодоступних інсталяцій Nomad у зв'язці з Consul. [95]

Порівняльно, Mesos історично орієнтований на шар спільного використання ресурсів із делегуванням політик фреймворкам (двоступеневе планування), що добре пасує до гетерогенних навантажень і кастомних планувальників; Nomad натомість надає єдину узгоджену площину планування з вбудованими багатоваріантними шедулерами й акцентом на простоті розгортання, мульти-дата-центровості та підтримці як контейнерних, так і «bare»-процесів. Обидва підходи застосовні для побудови відмовостійких розподілених систем, але відрізняються рівнем абстракції та складністю експлуатації: Mesos дозволяє об'єднати різні фреймворки в одному пулі (через «resource offers»), тоді як Nomad надає цілісний набір примітивів для сервісних, пакетних і системних задач у рамках одного оркестратора. Проте починаючи з серпня 2025 року проєкт Apache Mesos було заморожено [91].

2.3.6 Ansible

Ansible — альтернативний спосіб управління інфраструктурою, агентонезалежна (agentless) система автоматизації для керування конфігураціями, розгортання програм і оркестрації ІТ-процесів. Вона працює з «вузла керування» (control node), під'єднуючись до керованих вузлів стандартними протоколами: SSH для Unix-подібних систем та WinRM для Windows; за потреби модель «push» може бути інвертована у «pull» через утиліту ansible-pull. Такий підхід зменшує експлуатаційні витрати (немає агентів на вузлах), водночас зберігаючи керованість на масштабі кластера [97–100].

Базові абстракції Ansible — інвентар (inventory), плейбуки (playbooks) та модулі. Плейбуки у форматі YAML задають «бажаний стан» і послідовності завдань; значна частина модулів проєктована як ідемпотентна, тобто повторний запуск не змінює вже досягнутий стан цілі. Це спрощує багаторазове відтворюване розгортання складних застосунків [97–100].

Для повторного використання коду Ansible пропонує «ролі» (структуроване пакування змінних, файлів, завдань і хендлерів) та «колекції» (distribution-формат, що може включати модулі, плагіни, ролі й плейбуки). Поширення й інсталяція здійснюються через Ansible Galaxy та CLI-утиліту ansible-galaxy [97–100].

Інвентар може бути статичним (INI/YAML) або динамічним за допомогою інвентар-плагінів/скриптів, що підтягують списки хостів із хмарних та інших джерел. Рекомендовано застосовувати інвентар-плагіни нового покоління як основний механізм динамічного інвентарю [97–100].

Операційні можливості включають керування стратегіями виконання (linear, free) та поетапні оновлення через директиву serial, яка «б'є» множину хостів на батчі — типовий інструмент для rolling-upgrade без простоїв. Для

«сухих прогонів» і валідації впливу доступні режими `--check` (без внесення змін) і `--diff` (порівняння «до/після») [97–100].

Захист секретів реалізовано через Ansible Vault: можна шифрувати змінні й файли, керувати паролями сховища та додавати vault-ідентифікатори для багатоключових сценаріїв. Це дозволяє відокремити конфіденційні дані від артефактів розгортання та зберігати їх у VCS у зашифрованому вигляді [97–100].

Для централізованого керування на масштабі підприємства доступні надбудови: AWX (upstream-проект) надає веб-інтерфейс, REST API й рушій виконання поверх Ansible, тоді як Red Hat Ansible Automation Platform включає automation controller та інші компоненти для безпечного масштабування, делегування та керування життєвим циклом автоматизації. Окремо розвивається Event-Driven Ansible (rulebook-підхід), що запускає плейбуки/акції у відповідь на зовнішні події з моніторингових та інших систем [97–100].

З огляду на вищеописане та необхідність у відтворюваність змін — природно зводять до контейнеризації та оркестрації як до технічної основи запропонованого підходу. На цьому тлі Kubernetes виступає де-факто стандартом керування контейнеризованими навантаженнями у промислових середовищах, що забезпечує практичну застосовність методу в гетерогенних розподілених системах, де критичною є стійкість до каскадних відмов, спричинених несумісними змінами API. Крім цього, на відміну від Nomad, Kubernetes не вимагає ліцензування і фактично є відкритою платформою.

Таким чином, вибір Kubernetes обґрунтовується як з позицій наукової відповідності темі (API-орієнтована, декларативна модель керування станом), так і з позицій інженерної придатності (екосистема інструментів для комунікації, розгортання, спостережуваності й контролю сумісності), що

робить його найбільш прийнятною платформою для практичної реалізації методу суміснисно-керованого розгортання, окресленого у дисертації

2.4 Сервісна сітка як інфраструктурний шар сумісності

Сервісна сітка постає як спеціалізований інфраструктурний шар над прикладним рівнем, покликаний стандартизувати міжсервісну взаємодію в розподілених системах із мінімальним втручанням у прикладний код. Її місія — відокремити поперечні функції комунікації (адресація, безпека, надійність, спостережність) від бізнес-логіки, забезпечуючи єдині політики для різнорідних компонентів і, тим самим, зменшуючи імовірність несумісних змін API, що є центральною проблемою еволюції розподілених інтерфейсів. На тлі потреби у формалізованому контролі сумісності та керованій еволюції API такий стандартизований слой вносить інженерну дисципліну в обмін повідомленнями між сервісами та дозволяє реалізувати метод суміснисно-керованого розгортання на рівні експлуатації, а не лише процесів розроблення [101,102].

Безпечна комунікація в сервісній сітці реалізується через наскрізну автентифікацію й шифрування на транспортному рівні з взаємною перевіркою сторін. Криптографічна ідентичність сервісів і автоматизована ротація сертифікатів унеможливають «сірі» шляхи доступу, у яких клієнт обходить затверджені контракти взаємодії. У контексті еволюції API це означає, що доступ до нової або зміненої версії інтерфейсу регулюється не довірою до мережі, а явними політиками доступу, що зменшує ризик некерованих інтеграцій і перехресної несумісності між підсистемами. Таким чином, криптографічно закріплена ідентичність стає основою для реалізації політик

«допуску до версії», де несумісні комбінації просто не отримують каналного шляху [89,101,102].

Підвищення надійності комунікацій — ще одна ключова функція сервісної сітки. Ретрай із бюджетами, таймаути з відсіканням повільних шляхів, запобіжники і планомірна деградація навантаження перетворюють локальні збої постачальника API на керовані події. Ці механізми відокремлені від прикладного коду, тому їхню семантику можна уніфікувати для всіх сервісів. У підсумку знижується глибина розповсюдження збоїв і «радіус відмови», спричинених несумісними змінами інтерфейсів: навіть якщо нова версія API порушує контракт, наслідки локалізуються на рівні керуючих політик і не призводять до каскадних ефектів у клієнтських ланцюжках. Ця властивість узгоджується з дисертаційною моделлю каскадних відмов, де запроваджені метрики радіуса та глибини дозволяють інженерно оцінювати вплив несумісних змін і планувати обмеження їхнього поширення [89,101,102].

Спостережність трафіку у сервісній сітці забезпечує телеметрію запитів, трасування розподілених транзакцій та агреговані метрики без модифікації програмного коду. Це створює «зовнішній» канал верифікації контрактної поведінки: відхилення схем даних, атипові коди відповіді чи неспівмірні латентності стають сигналами про можливе порушення сумісності в момент появи, а не постфактум. Наявність єдиної моделі даних спостереження для всіх сервісів дозволяє автоматизувати виявлення регресій, пов'язаних із версіонуванням API, і під'єднати такі сигнали до процедур прийняття рішень щодо розгортання або ізоляції конкретних версій [89,101,102].

2.5 GitOps як стандарт експлуатації керованої еволюції

GitOps — це операційна парадигма, у якій бажаний стан інфраструктури й застосунків описується декларативно та зберігається в системі контролю версій, а відповідність реального стану бажаному забезпечується автоматизованим циклом узгодження. На відміну від імперативного керування середовищем, GitOps трактує розгортання як відтворену проєкцію коду конфігурації, забезпечуючи детермінізм, аудитованість і оперативне відновлення. Для розподілених систем, де інтерфейси змінюються асинхронно, це принципово важливо: саме інваріанти сумісності API переносяться в «простір коду» і стають об'єктами того ж життєвого циклу, що й функціональні зміни [89,103].

Фундамент GitOps становить оголошувальність: кожен релевантний аспект кластера — від описів сервісів до політик доступу та матриць сумісності між версіями — заданий як кодовий артефакт. Це на практиці означає, що узгодженість протоколів взаємодії, обмеження для перехресних викликів і «правила переходу» між версіями API можуть бути формалізовані у вигляді декларативних специфікацій, а не усних домовленостей або розрізнених скриптів. Декларативність відкриває шлях до автоматичних перевірок: зміни до таких специфікацій проходять ті самі чекапи, що і прикладний код, і не можуть бути застосовані, допоки не підтверджена відповідність вимогам сумісності [89,103,104].

Інший принцип — «єдине джерело істини» — піднімає репозиторій конфігурації до статусу операційного контракту між командами розроблення та експлуатації. Саме в ньому фіксуються вимоги до версіонування, інваріанти сумісності та допустимі комбінації компонентів. Це критично з погляду керованої еволюції API: з'являється однозначний, версіонований опис системи, до якого можна застосовувати формальні політики прийняття змін. У

поєднанні з інструментами перевірок та рецензій така організація різко знижує ризик неузгодженого «просочення» несумісних змін у середовище виконання. У термінах дисертації це практичний механізм «блокування несумісних релізів», який діє до моменту безпосереднього розгортання [89,103,104].

Автоматизований цикл узгодження забезпечує безперервне застосування описаного у сховищі бажаного стану до реального середовища та симетрично — виявлення і корекцію дрейфу конфігурації. Перевага такого підходу в контексті сумісності полягає в тому, що будь-яке відхилення від затверджених версійних поєднань, політик маршрутизації чи схем даних буде виявлене як відхилення від коду, а не як «особливість» конкретного стенду. Завдяки цьому відновлюваність середовища стає неопераційною опцією, а очікуваною властивістю: повернення до відомої сумісної конфігурації — це звичайна операція відкату до перевіреного коміту [89,103,104].

Зменшення дрейфу конфігурації є прямим наслідком постійної конвергенції до бажаного стану. У практиці це проявляється у відсутності «сюрпризів» під час приймальних випробувань та у стабільності відтворення тестових сценаріїв для сумісності API. Замість множини відмінних стендів, кожен із яких живе за власними неписаними правилами, GitOps створює умови для відтворюваної регресії: якщо комбінація версій пройшла перевірки та була задокументована, її поведінка буде однаковою на будь-якому етапі шляху до продуктиву. Це безпосередньо підтримує заявлену в дисертації мету — вибір і підтримання сумісних конфігурацій у присутності несумісних змін [89,103–105].

Взаємодія GitOps із сервісною сіткою формує замкнений контур керування сумісністю. З одного боку, GitOps надає декларативний «носій волі» — політики маршрутизації між версіями, обмеження доступу, схеми даних і матриці узгодженості, а також процедури їхнього поступового введення. З

іншого — сервісна сітка виконує ці політики та постачає телеметрію фактичної поведінки, що повертається у процес ухвалення змін. Якщо телеметрія сигналізує про порушення контрактної поведінки або про зростання радіуса відмови, GitOps-цикл фіксує «червоний» стан і блокує подальшу промоцію версії або ініціює відкат до останньої відомо-сумісної конфігурації. Така зв'язка безпосередньо реалізує постулат дисертації про автоматизоване блокування несумісних релізів і кероване згортання їхнього впливу на екосистему сервісів [89,103–105].

Висновки до розділу

У розділі проведено системний аналіз архітектур розподілених програмних систем, що підтвердив різноманітність підходів до організації взаємодії сервісів (подійно-орієнтовані, сервіс-орієнтовані, керовані оркестратором, просторові та ін.). Це дало змогу сформулювати концептуальне підґрунтя для побудови моделі середовища, у якому виникають ризики несумісності API.

Встановлено, що при сучасних процесах розгортання в різних середовищах інтеграція методу виявлення несумісних версій сервісів перед розгортанням розподілених програмних систем дозволить автоматично ідентифікувати конфлікти між версіями сервісів, запобігаючи утворенню некоректних конфігурацій у робочому середовищі.

У межах дослідження розглянуто різні варіанти середовищ для реалізації методу та обґрунтовано вибір Kubernetes як оптимального рішення. Його оркестраційні можливості забезпечують масштабованість, контроль життєвого циклу сервісів і гнучке керування оновленнями, що дозволяє ефективно інтегрувати запропонований метод у процеси розгортання.

Таким чином, розділ закладає науково-методичну основу для подальшої програмної реалізації розробленого методу та експериментальної перевірки його ефективності.

РОЗДІЛ 3. ПРОГРАМНА РЕАЛІЗАЦІЯ ТА АРХІТЕКТУРА ЕКСПЕРИМЕНТАЛЬНОГО СТЕНДУ

3.1 Опис методу

Основною ідеєю методу є недопущення розміщення на робочому середовищі компонентів інформаційної системи, які мають несумісні API, тим самим усуваючи проблему невідповідності API до її можливої появи.

Ядром методу є аплікація під назвою «Реєстр сумісності API». Ця аплікація зберігає в собі список сервісів із версіями, та їх API з версіями, і будує необхідні дані для сценарію розгортання з урахуванням сумісності програмних інтерфейсів розподілених систем. У системах з розподіленою архітектурою прикладні API міжкомпонентної комунікації можуть автоматично генеруватися з версіонованих спільних ресурсів, наприклад зі схем JSON, Proto2/3. Система Continuous Integration (CI) далі фіксує назву сервісу та версію API у *Реєстрі сумісності API* (Рисунок). Отримує відповідь про потенційну наявність поєднаних за API сервісів для сповіщення розробників. А також формує сценарій розміщення найсвіжіших сервісів тільки із сумісними версіями API.

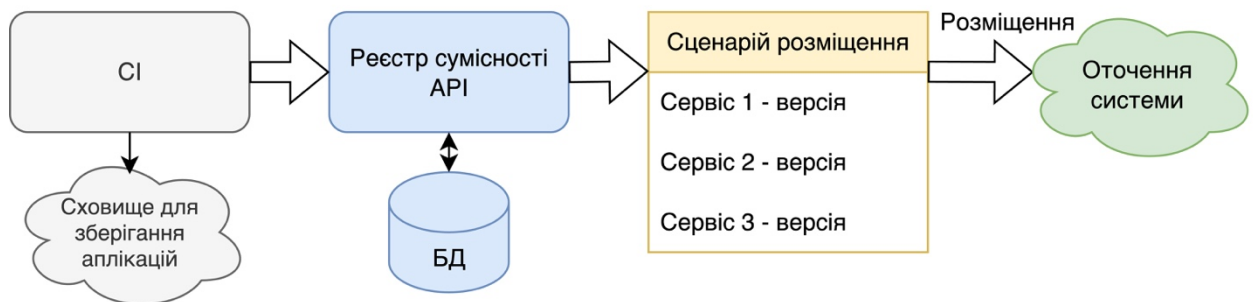


Рисунок 3.1 Оркестрація компонентів розподілених систем на основі сумісності з API.

Реєстр сумісності API полегшує керування змінами API та дає змогу автоматично відстежувати актуальність версій для всіх інформаційних сервісів. На відміну від традиційних підходів до версіонування, що переважно зосереджені на підтриманні історії документації API, запропонований метод вводить механізм оркестрації, який активно керує сумісністю версій до розгортання. Така оркестрація гарантує дотримання інформаційними сервісами вимог сумісності, забезпечуючи надійний механізм підтримання узгодженості та стабільності в динамічних системах. Підхід мінімізує ризики, пов'язані з оновленнями API, і уможливлює безшовну інтеграцію та координацію між інформаційними сервісами.

Для операціоналізації оркестрації версій інформаційних сервісів у рамках платформи інтегруються специфічні для протоколів засоби валідації сумісності. Наприклад, для сервісів на основі gRPC використовується контейнеризована версія утиліти `protolock` [106] з метою виявлення несумісних зі зворотною сумісністю змін у визначеннях інтерфейсів `.proto`. Цей інструмент ідентифікує несумісні модифікації сигнатур сервісів, зокрема вилучення полів або несумісні зміни типів, до того, як розгортання продовжиться.

Для протоколів, описаних JSON Schema, у межах платформи застосовується засіб IBM `jsonschema`, описаний у [107], що реалізує порівняння на основі підтипів. У цьому підході, якщо нова схема є підтипом попередньої, зміна класифікується як сумісна і спричиняє інкремент молодшого номера версії. Навпаки, якщо попередня схема є підтипом нової (але не навпаки), виконується інкремент старшого номера через потенційну несумісність. Валідатор підтримує вкладені структури даних (наприклад, DTO або JSON-об'єкти з багатьма полями), щоб виявляти семантичні зміни поза межами суто синтаксичних сигнатур. Це суттєво, оскільки структури

параметрів можуть містити кілька частин даних, значення яких може змінюватися навіть за незмінної сигнатури методу.

Версія API має складатися з двох чисел, розділених крапкою [108]. Перше число позначає внесення несумісних змін відносно попередньої версії, друге — незначні оновлення. Такий підхід до версіонування забезпечує контроль точності розгортання компонента інформаційного сервісу, поєднуючи стабільність взаємодії між сервісами з поетапним упровадженням нових функцій.

Реєстр сумісності API використовує надані дані про версії для формування сценаріїв розгортання на основі сумісності версій API між провайдерами та споживачами. Із підмножини версій сервісів, що відповідають задекларованим критеріям сумісності, застосунок автоматично обирає останній випуск (тобто найбільше семантичне значення версії), аби розгортання завжди виконувалося на найактуальнішій, але водночас сумісній реалізації. Запропонований метод автоматизує цей процес, програмно перевіряючи залежності та сповіщаючи зацікавлені сторони у випадках, коли зміни потребують втручання.

Такий підхід, керований автоматизацією, гарантує, що розробники отримують практичні результати, а не лише «сиру» історію версій, спрощуючи імплементацію змін API і зменшуючи тертя під час розгортання. Процес підтримує баланс між збереженням стабільності системи та сприянням інноваціям у контрольований спосіб (Рисунок).

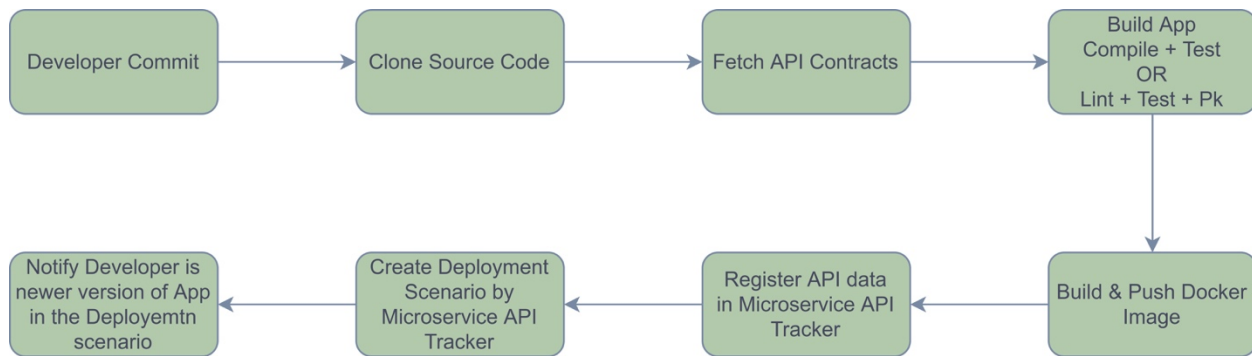


Рисунок 3.2 Блок-схема CI

Запропонований підхід є оптимальним для систем, де операційна стабільність є критично важливою [109]. Наприклад, у системах управління повітряним рухом [110,111] або інфраструктурах управління енергомережею [1,2,4,112] навіть незначні збої можуть призвести до значних фінансових втрат, загроз безпеці людей або порушень громадської безпеки. Однак через складність проектування, управління та обслуговування, використання «Метод розгортання інформаційних сервісів з урахуванням сумісності програмних інтерфейсів розподілених систем» може бути менш застосовним для систем, де стабільність не є першочерговою проблемою. У таких контекстах, як невеликі веб-сайти, внутрішні корпоративні програми або експериментальні платформи з низьким навантаженням, акцент може зміститися на швидкість розробки та зниження витрат, оскільки потенційні збої в таких системах зазвичай мають мінімальні наслідки. Таким чином, хоча цей підхід забезпечує значні переваги для критично важливих систем, його складність може переважувати його переваги в середовищах, що менш залежать від стабільності.

Цей підхід мінімізує збої, поєднуючи перевірку сумісності з прогресивними стратегіями зміни трафіку, такими як розгортання Blue/Green. Після розгортання та перевірки нової версії трафік поступово

перенаправляється зі стабільного (синього) середовища до нового (зеленого), що гарантує миттєве та безризикове повернення до попередньої версії у разі неочікуваних збоїв. Це забезпечує контрольований процес розгортання з мінімальним часом простою, видимим для користувача, підвищуючи надійність взаємодії між інформаційними сервісами. Автоматизоване управління версіями API та їх сумісністю гарантує стабільність системи навіть у складних архітектурах з численними залежностями. Тим не менш, ця стратегія вимагає координації майже одночасного розгортання кількох інформаційних сервісів та достатніх інфраструктурних ресурсів для підтримки паралельних середовищ, що може збільшити складність впровадження.

Для зберігання інформації про версії сервісів створена таблиця в SQL базі даних Postgres (Рисунок). Базу даних PostgreSQL обрано, оскільки вона забезпечує відповідність міжнародним стандартам SQL. Крім того, як система з відкритим кодом, PostgreSQL поєднує масштабованість і відмовостійкість, що робить її оптимальною для використання у критично важливих розподілених системах. Також використання бази не вимагає додаткових витрат на ліцензії [113].

SERVICE_API_VERSIONS		
BIGINT	id	PRIMARY KEY, identity
VARCHAR	service_id	NOT NULL
VARCHAR	service_name	NOT NULL
VARCHAR	api_id	NOT NULL
VARCHAR	api_type	NOT NULL
INT	api_version_major	NOT NULL
INT	api_version_minor	NOT NULL
INT	service_version_major	NOT NULL
INT	service_version_minor	NOT NULL
TIMESTAMP	created_at	NOT NULL, DEFAULT now()

Рисунок 3.3. Таблиця для зберігання версій сервісів

Для отримання з таблиці списку сервісів, які сумісні за API використовується наступний SQL запит:

```
WITH api_version_groups AS (
  SELECT
    api_id,
    api_type,
    api_version_major,
    api_version_minor,
    COUNT(DISTINCT service_id) as service_count
  FROM service_api_versions
  GROUP BY api_id, api_type, api_version_major, api_version_minor
),
most_common_api_versions AS (
```

```

SELECT
    api_id,
    api_type,
    api_version_major,
    api_version_minor
FROM (
    SELECT
        *,
        ROW_NUMBER() OVER (
            PARTITION BY api_id, api_type
            ORDER BY service_count DESC, api_version_major DESC,
api_version_minor DESC
        ) as rn
    FROM api_version_groups
) ranked
WHERE rn = 1
),
service_data AS (
    SELECT
        s.service_id,
        s.service_name,
        s.service_version_major,
        s.service_version_minor,
        json_agg(
            json_build_object(
                'api_id', s.api_id,

```



```

        'api_type', s.api_type,
        'api_version_major', m.api_version_major,
        'api_version_minor', m.api_version_minor
    )
) as apis
FROM service_api_versions s
JOIN most_common_api_versions m ON s.api_id = m.api_id AND s.api_type =
m.api_type
GROUP BY s.service_id, s.service_name, s.service_version_major,
s.service_version_minor
),
latest_services AS (
    SELECT *
    FROM (
        SELECT *,
        ROW_NUMBER() OVER (
            PARTITION BY service_id
            ORDER BY service_version_major DESC, service_version_minor DESC
        ) as rn
        FROM service_data
    ) ranked_services
    WHERE rn = 1
)
SELECT
    service_id,
    service_name,

```

```
service_version_major,  
service_version_minor,  
apis  
FROM latest_services  
ORDER BY service_name;
```

3.2 Поєднання з процесуальним підходом

Запропонований метод виявлення несумісних версій сервісів органічно поєднується з описаним вище процесуальним підходом, який передбачає перевірку нового програмного забезпечення у проміжному середовищі, максимально наближеному до робочого

Метод забезпечує проактивну перевірку версій API ще до розгортання у тестове-середовище. Це означає, що до середовища потрапляють лише ті комбінації сервісів, які пройшли автоматизовану перевірку на сумісність. Таким чином, процесуальний підхід використовується не як фільтр для «сирих» версій, а як інструмент валідації вже попередньо узгоджених компонентів. Це суттєво знижує навантаження на спеціалістів з перевірки якості та скорочує витрати часу на локалізацію помилок.

Крім того, процесуальний підхід може бути розширений для імітації каскадних залежностей між сервісами, що дозволяє оцінити не лише технічну сумісність API, але й поведінку системи в умовах реального навантаження. У поєднанні з методом це створює багаторівневий механізм контролю: на першому рівні автоматично блокується несумісний реліз, а на другому — у процесуальному підході перевіряється відповідність системи функціональним вимогам.

Таким чином, інтеграція методу в процесуальний підхід формує двоетапний бар'єр проти несумісностей: спочатку на рівні аналізу API, а потім у тестовому середовищі, що гарантує більш високий рівень стабільності при подальшому переході до продуктивного середовища.

Висновки до розділу

У цьому розділі було реалізовано архітектурний та програмний підхід до створення експериментального стенду, що забезпечує відтворення умов реального функціонування розподілених систем.

Розроблено програмне забезпечення для перевірки ефективності методу, що дозволило здійснити його експериментальну апробацію на архітектурно-програмному стенді та підтвердити придатність до практичного застосування у реальних сценаріях розподілених систем. Реалізовані інструменти забезпечують автоматизовану перевірку сумісності gRPC- та JSON Schema-сервісів та перемикання трафіку на новий реліз інтегруються у конвеєри CI/CD, що дозволило обґрунтувати підвищення надійності і відмовостійкості при оновленні критично важливих інформаційних сервісів.

Дістали подальшого розвитку існуючі стратегії мінімізації впливу несумісних змін API у розподілених системах за рахунок їх комбінації із запропонованим методом, а саме поєднання описаного методу з процесуальним підходом. Така інтеграція забезпечує багаторівневу перевірку стабільності: на першому рівні відбувається автоматизоване блокування несумісних версій API ще до моменту їхнього потрапляння у тестове середовище, а на другому — здійснюється відтворення повноцінних сценаріїв роботи системи в умовах тестового середовища. Це дозволяє не лише

гарантувати технічну сумісність інтерфейсів, але й оцінити поведінку сервісів під реалістичними навантаженнями.

РОЗДІЛ 4. ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ТА ВЕРИФІКАЦІЯ ЕФЕКТИВНОСТІ МЕТОДУ

4.1 Постановка експерименту та методологія

Експериментальне дослідження було спрямоване на оцінювання стійкості автоматизованого фреймворку оркестрації версій інформаційних сервісів за умов виникнення несумісних змін у визначеннях JSON Schema та gRPC. Ключовою метою було виявлення порушень сумісності та перевірка здатності системи координувати безпечні розгортання на підставі версійних обмежень.

Постановка експерименту: кластер Kubernetes на базі MicroK8s версії 1.31.7 (ревізія 7979), розгорнутий на двох одноплатних комп'ютерах OrangePi 5, кожен із 16 ГБ оперативної пам'яті та 64 ГБ вбудованого сховища, з процесорами Rockchip RK3588 (Фучжоу, КНР) (Рисунок). Системне навантаження імітувалося сценарієм Gatling, налаштованим на сталу інтенсивність 100 HTTP-запитів за секунду протягом 10 хв.

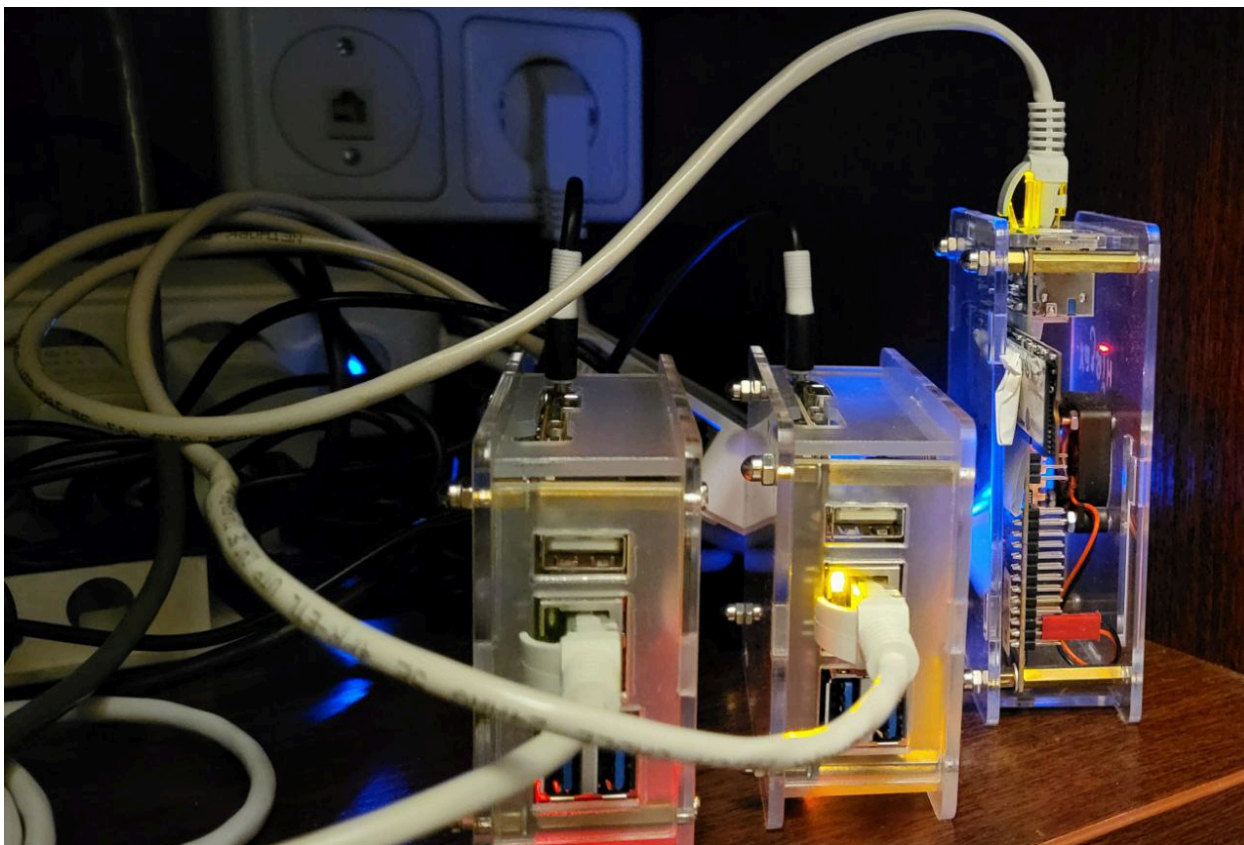


Рисунок 4.1 Фото частини лабораторного стенду

Досліджувана система має послідовну топологію з трьох компонентів:
вхідні HTTP-запити → сервіс1 → сервіс2 → сервіс3.

Комунікаційні протоколи:

- сервіс1 → сервіс2: gRPC (бінарний протокол, що описується в текстовому файлі)
- сервіс2 → сервіс3: HTTP з використанням JSON Schema для формування коду та валідації.

Визначення інтерфейсів API зберігаються в окремих репозиторіях Git і інтегруються до кожного інформаційного сервісу як підпроекти. Конвеєр безперервної інтеграції Jenkins відстежує зміни у визначеннях протоколів і виконує такі кроки:

1. перевірка зворотної сумісності;
2. застосування семантичного версіонування:
 - мажорне підвищення версії у разі несумісних змін;
 - міnorне підвищення версії для сумісних, сумісних змін;
3. оновлення файлу version.info та призначення відповідних Git-тегів.

Весь код доступний публічно на GitLab за посиланням <https://gitlab.com/frameork-orchestartor-api-versions>

4.2 Контрольоване внесення несумісної зміни

Було внесено несумісну модифікацію до JSON-схеми, що регламентує протокол взаємодії між сервіс2 та сервіс3, а саме — змінено назву одного з полів. Після фіксації (commit) цієї зміни конвеєр CI Jenkins автоматично ідентифікував зворотну несумісність і виконав мажорне підвищення версії. Відповідно до оновленої версії залежний інформаційний сервіс було модифіковано для узгодження з переглянutoю схемою.

Для суворої оцінки наслідків зазначеної зміни проведено навантажувальне тестування за допомогою Gatling із послідовною маршрутизацією запитів крізь ланцюг інформаційних сервісів сервіс1 → сервіс2 → сервіс3 (Рисунок). У процесі випробувань оновлений сервіс сервіс2 було вручну розгорнуто в кластері Kubernetes для імітації продукційного середовища. Унаслідок цього Gatling (Рисунок 4) зафіксував низку HTTP-помилки, що походили від MS3 та були зумовлені відмовами валідації схеми через відсутність очікуваного поля. Отримані результати підтверджують, що внесена модифікація JSON-схеми породила несумісність між сервіс2 і сервіс3.

Крім того, спостережуваний радіус відмов $r = 2$ узгоджується з прогнозом рівняння (3): відмова в сервіс2 поширилася «далі за потоком» на сервіс3 і, своєю чергою, вплинула на зовнішнього клієнта.



Рисунок 4.2 Кероване перемикання трафіку під час зміни версії сервісу в Kubernetes

— доступність розгортання, — успішні запити, — неуспішні запити

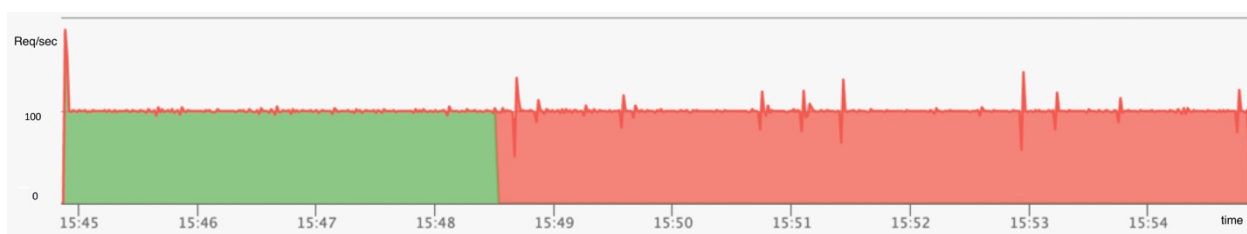


Рисунок 4 Вплив оновлення версії вузлів на частку успішних запитів у розгортанні Kubernetes

— успішні запити, — неуспішні запити

4.3 Blue/Green-розгортання за наявності несумісної схеми

У цьому сценарії було застосовано стратегію Blue/Green-розгортання, причому середовище Green містило версію інформаційного сервісу сервіс2 з

несумісною зміною в JSON-схемі. О 20:58 запроваджено версію Green; у проміжку 20:58–20:59 відбулося поетапне переключення трафіку на середовище Green. До 21:01 вбудовані механізми захисту Kubernetes ініціювали дроселювання трафіку та circuit breaking унаслідок збоїв запитів, спричинених невідповідністю схем (Рисунок 5). Відтак система Gatling почала отримувати виключно неуспішні відповіді (Рисунок 6).



Рисунок 5 Перемикання трафіку під час зміни версії сервісу в Kubernetes

— доступність розгортання Blue, — доступність розгортання Green,
— успішні запити, — неуспішні запити

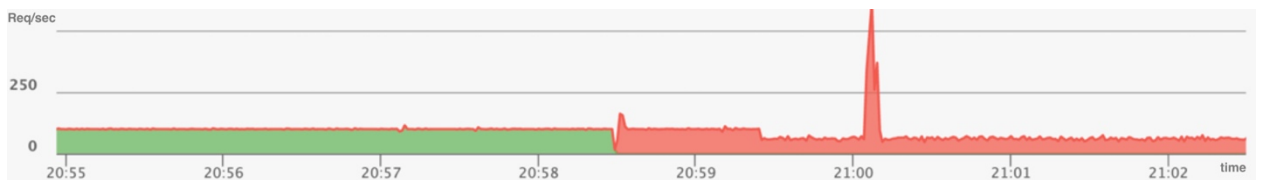


Рисунок 6 Вплив перемикання трафіку зі стабільного синього розгортання на зелений

— успішні запити, — неуспішні запити

Отримані результати підтвердили нестабільність функціонування під час виконання, зумовлену несумісністю схеми під навантаженням, попри часткову ізоляцію, яку забезпечує підхід Blue/Green-розгортання.

4.4 Скординоване розгортання оновлених сумісних версій замість існуючих

У цьому сценарії інформаційний сервіс сервіс3 було оновлено для підтримки новішої версії JSON-схеми, яка вже використовується сервіс2. *Реєстр сумісності API* згенерував узгоджену послідовність розгортання для сервіс1, сервіс2 та сервіс3, у межах якої для сервіс2 і сервіс3 передбачено нові версії; цю послідовність було виконано скординовано.

Попри те, що всі інформаційні сервіси оновлено до сумісних версій, під час розгортання під навантаженням зафіксовано системні проблеми, зумовлені внесенням несумісних змін до сервісних API. Ключова причина полягає в неоднорідності часових характеристик оновлення: окремі сервіси завантажуються та запускаються швидше за інші. Унаслідок цього виникає перехідний стан, коли старі й нові версії одночасно активні, що призводить до тимчасових невідповідностей контрактів API між взаємодіючими компонентами (Рисунок 7).



Рисунок 7 Перемикання трафіку під час зміни версій сервісів в Kubernetes

— доступність розгортання Blue, — успішні запити, — неуспішні запити

Ця неузгодженість породжує умову гонки (race condition), за якої окремі сервіси можуть надсилати або приймати запити, несумісні з поточним станом їхніх пірів, що призводить до відмов запитів або непередбачуваної поведінки. Явище особливо загострюється під навантаженням, коли системний стрес посилює часові розбіжності між моментами запуску та готовності сервісів (Рисунок 8).

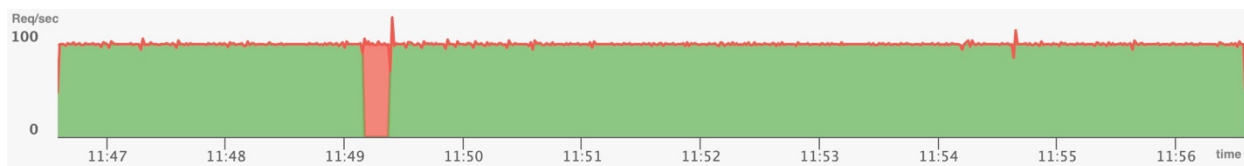


Рисунок 8 Вплив оновлення версії вузлів на коефіцієнт успішності запитів у кластері Kubernetes

— успішні запити, — неуспішні запити

Як видно з моніторингових графіків, приблизно о 11:49 зафіксовано сплеск HTTP-помилки, що збігся у часі з розгортанням оновлених версій

інформаційних сервісів. Ці помилки тривали близько однієї хвилини та свідчать про тимчасову несумісність API між інформаційними сервісами. Зокрема, дані вказують, що в зазначений інтервал частина сервісів і надалі працювала зі старою версією API, несумісною з щойно розгорнутими версіями.

4.5 Метод виявлення несумісних версій сервісів перед розгортанням розподілених програмних систем

Для верифікації гіпотези та мінімізації проблем невідповідності API, зафіксованих під час поступових оновлень (rolling updates) під навантаженням, попередній експеримент було повторено в повному варіанті: із застосуванням стратегії переключення трафіку. У цій конфігурації нову версію інформаційних сервісів (середовище green) повністю розгорнули паралельно до наявної продукційної версії (середовище blue). Маршрутизацію трафіку на оновлену версію виконано лише після підтвердження, що всі green-сервіси є справними (healthy) та повністю працездатними.

Розгортання нової версії о 15:09 супроводжувалося переключенням трафіку до 15:10, без фіксації помилок чи спрацювань механізмів circuit breaking (Рисунок 9). Під час перехідного інтервалу спостерігалось незначне зростання затримки відповіді, що узгоджується з очікуваною поведінкою системи під навантаженням (Рисунок 10).

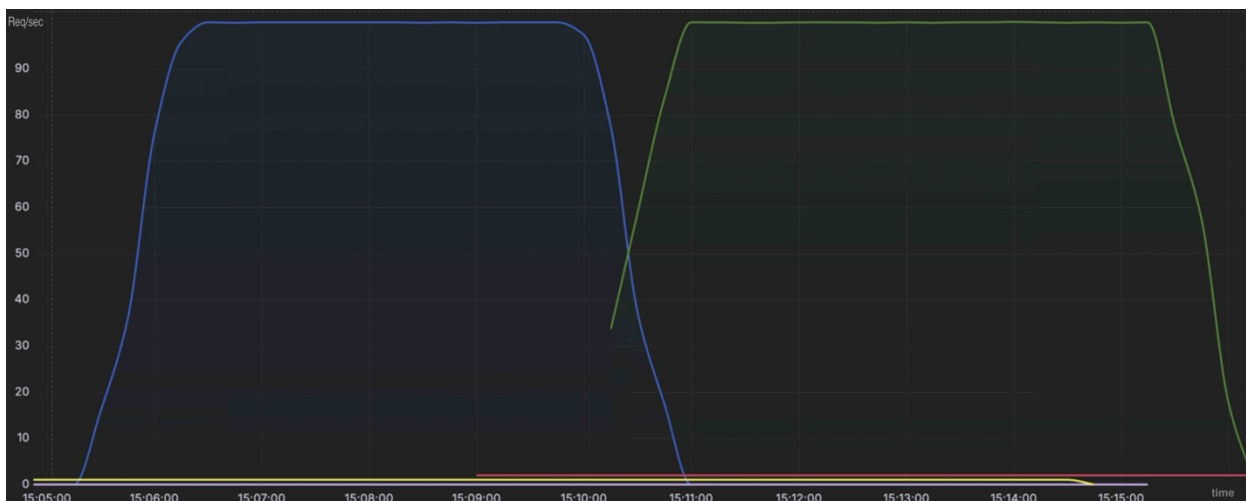


Рисунок 9 Кероване переключення трафіку під час зміни версії сервісу в Kubernetes

— доступність розгортання Blue, — доступність розгортання Green,
— успішні запити розгортання Blue, — успішні запити розгортання Green

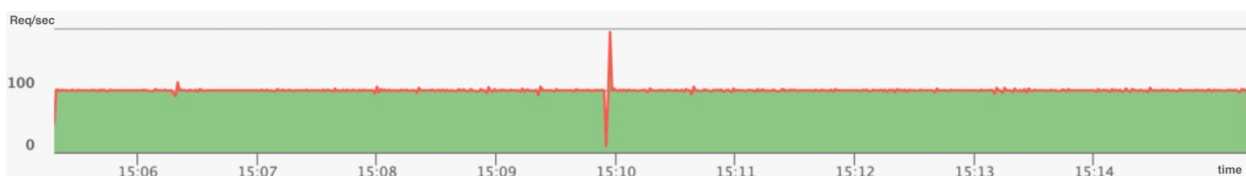


Рисунок 10 Ефект оновлення версії інформаційних сервісів на успішність запитів у кластері Kubernetes

— успішні запити, — неуспішні запити

Висновки до розділу

Експериментальне дослідження підтвердило ефективність запропонованого методу виявлення несумісних версій сервісів перед розгортанням. Було реалізовано інструментальну підтримку для перевірки сумісності gRPC- та JSON Schema-сервісів. У процесі контрольованого внесення несумісних змін було показано, що система здатна своєчасно блокувати небезпечні релізи та запобігати каскадним відмовам. Використання

стратегії перемикавання трафіку дозволило оцінити робастність методу, а сценарії скоординованого оновлення сервісів довели його придатність для систем із високою частотою змін. Результати експериментів засвідчили, що інтеграція запропонованого підходу у конвеєри CI/CD підвищує рівень надійності та стабільності розподілених програмних систем, знижуючи ризики збоїв на етапі еволюції API, які призводять до появи несумісних змін при міжсервісній комунікації.

ВИСНОВКИ

У дисертаційній роботі наведено вирішення актуальної наукової задачі підвищення надійності та стабільності розподілених програмних систем за рахунок розроблення та впровадження методу виявлення несумісних версій сервісів перед їх розгортанням. Основні наукові та практичні результати роботи полягають у наступному:

1. Обґрунтовано технологічні особливості еволюції API у розподілених архітектурах, яка обумовлена потребами в оновленні функціональності, технологічній адаптації та вдосконаленні дизайну. Визначено, що асинхронна природа цих змін становить ключовий ризик, спричиняючи несумісні зміни (breaking changes), які порушують зворотну сумісність і можуть призводити до неконтрольованої зупинки важливих вузлів. Підтверджено, що ефект ланцюгової реакції від несумісних змін API може спричиняти каскадні відмови та простої сервісів у критично важливих доменах, таких як енергетика, фінтех та авіоніка. Це підтверджує, що ефективне управління версіями та контроль сумісності є не просто бажаним, а необхідною умовою підтримання працездатності та керованості еволюції розподілених систем. Таким чином, актуальним науковим завданням є розроблення метод виявлення несумісних версій сервісів перед розгортанням розподілених програмних систем.
2. Проведений порівняльний аналіз існуючих стратегій мінімізації впливу несумісних змін виявив їхні суттєві обмеження. Встановлено, що більшість цих підходів є реактивними, виявляючи несумісність лише на етапі тестування або розгортання, або ж вимагають значних інфраструктурних витрат чи призводять до накопичення технічного боргу.

3. Дістали подальшого розвитку існуючі стратегії мінімізації впливу у розподілених системах за рахунок їх комбінації із запропонованим методом, що дозволило обґрунтувати поєднання з процесуальним підходом. Така інтеграція забезпечує багаторівневу перевірку стабільності: на першому рівні відбувається автоматизоване блокування несумісних версій API ще до моменту їхнього потрапляння у тестове середовище, а на другому — здійснюється відтворення повноцінних сценаріїв роботи системи в умовах тестового середовища. Це дозволяє не лише гарантувати технічну сумісність інтерфейсів, але й оцінити поведінку сервісів під навантаженнями, наближеними до продуктивних.
4. Розроблено метод виявлення несумісних версій сервісів перед розгортанням розподілених програмних систем, який, на відміну від існуючих, базується на формалізованих критеріях сумісності API, що забезпечує автоматизацію блокування несумісних релізів, знижує ризики каскадних відмов через несумісні зміни API і гарантує вибір сумісних конфігурацій не допускаючи появи несумісних API в робочому середовищі.
5. Дістали подальшого розвитку існуючі стратегії мінімізації впливу несумісних змін API у розподілених системах за рахунок їх комбінації із запропонованим методом, а саме поєднання описаного методу з процесуальним підходом. Така інтеграція забезпечує багаторівневу перевірку стабільності: на першому рівні відбувається автоматизоване блокування несумісних версій API ще до моменту їхнього потрапляння у тестове середовище, а на другому — здійснюється відтворення повноцінних сценаріїв роботи системи в умовах тестового середовища. Це дозволяє не лише гарантувати технічну сумісність інтерфейсів, але й оцінити поведінку сервісів під очікуваними навантаженнями.

6. Розроблено програмне забезпечення для перевірки ефективності методу, що дозволило здійснити його експериментальну апробацію на архітектурно-програмному стенді та підтвердити придатність до практичного застосування у реальних сценаріях розподілених систем. Реалізовані інструменти забезпечують автоматизовану перевірку сумісності gRPC- та JSON Schema-сервісів та перемикання трафіку на новий реліз інтегруються у конвеєри CI/CD, що дозволило обґрунтувати підвищення надійності і відмовостійкості при оновленні критично важливих інформаційних сервісів.
7. Експериментальне дослідження підтвердило ефективність запропонованого методу виявлення несумісних версій сервісів перед розгортанням. У процесі контрольованого внесення несумісних змін було показано, що система здатна своєчасно блокувати небезпечні релізи та запобігати каскадним відмовам. Використання стратегії перемикання трафіку дозволило оцінити робастність методу, а сценарії скоординованого оновлення сервісів довели його придатність для систем із високою частотою змін. Загалом результати експериментів засвідчили, що інтеграція запропонованого підходу у конвеєри CI/CD підвищує рівень надійності та стабільності розподілених програмних систем, знижуючи ризики збоїв на етапі їхньої еволюції.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Yaroshynskyi M., Prymushko A., Puchko I., Sirotkin O., Sinko D. Akka as a tool for modelling and managing a smart grid system. *Journal of Edge Computing*. 2025. Vol. 4, No. 1. P. 105–115. DOI: 10.55056/jec.822.
2. Prymushko A., Puchko I., Yaroshynskyi M., Sinko D., Kravtsov H., Artemchuk V. Efficient State Synchronization in Distributed Electrical Grid Systems Using Conflict-Free Replicated Data Types. *IoT. Multidisciplinary Digital Publishing Institute*, 2025. Vol. 6, No. 1. P. 6. DOI: 10.3390/iot6010006.
3. А.М. Примушко, І.В. Пучко, М.С. Ярошинський, Д.П. Сінько. Програмний Дизайн Розподіленої Високонавантаженої Системи Електроенергетичної Мережі На Базі Моделі Акторів Із Застосуванням Смарт-Контрактів. *Електронне моделювання*. 2024. Vol. 46, No. 3. P. 57–72. DOI: 10.15407/emodel.46.03.057.
4. Prymushko A., Yaroshynskyi M., Puchko I. Representation and Synchronization of States of Distributed Electrical Grid Systems Based on Conflict Free Replicated Data Types. *2024 14th International Conference on Dependable Systems, Services and Technologies (DESSERT): 2024 14th International Conference on Dependable Systems, Services and Technologies (DESSERT)*. 2024. P. 1–5. DOI: 10.1109/DESSERT65323.2024.11122143.
5. Yaroshynskyi M., Puchko I., Prymushko A., Kravtsov H., Artemchuk V. Investigating the Evolution of Resilient Microservice Architectures: A Compatibility-Driven Version Orchestration Approach. *Digital. Multidisciplinary Digital Publishing Institute*, 2025. Vol. 5, No. 3. P. 27. DOI: 10.3390/digital5030027.
6. Ярошинський М.С., Пучко І.В. Способи Розв’язання Проблеми Асинхронності Зміни Прикладного Програмного Інтерфейса в

- Мікросервісній Архітектурі. *Електронне моделювання*. 2025. Vol. 47, No. 4. P. 57–72. DOI: 10.15407/emodel.47.04.057.
7. Practical API Design. Berkeley, CA: Apress, 2008. DOI: 10.1007/978-1-4302-0974-4.
 8. Lercher A., Glock J., Macho C., Pinzger M. Microservice API Evolution in Practice: A Study on Strategies and Challenges. *Journal of Systems and Software*. 2024. Vol. 215. P. 112110. DOI: 10.1016/j.jss.2024.112110.
 9. Dig D., Johnson R. How Do APIs Evolve? A Story of Refactoring: Research Articles. *J. Softw. Maint. Evol.* 2006. Vol. 18, No. 2. P. 83–107.
 10. Sirotkin O., Prymushko A., Puchko I., Kravtsov H., Yaroshynskyi M., Artemchuk V. Parallel Simulation Using Reactive Streams: Graph-Based Approach for Dynamic Modeling and Optimization. *Computation*. Multidisciplinary Digital Publishing Institute, 2025. Vol. 13, No. 5. P. 103. DOI: 10.3390/computation13050103.
 11. Brito A., Valente M.T., Xavier L., Hora A. You broke my code: understanding the motivations for breaking changes in APIs. *Empirical Software Engineering*. 2020. Vol. 25, No. 2. P. 1458–1492. DOI: 10.1007/s10664-019-09756-z.
 12. Gu H., He H., Zhou M. Self-Admitted Library Migrations in Java, JavaScript, and Python Packaging Ecosystems: A Comparative Study. *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*: 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). 2023. P. 627–638. DOI: 10.1109/SANER56733.2023.00064.
 13. Yılmaz R., Buzluca F. A Fuzzy Logic-Based Quality Model for Identifying Microservices with Low Maintainability. *Journal of Systems and Software*. 2024. Vol. 216. P. 112143. DOI: 10.1016/j.jss.2024.112143.

14. Microservices [Электронный ресурс]. *martinfowler.com*. URL: <https://martinfowler.com/articles/microservices.html> (Режим доступа: 13.07.2025).
15. Mondal S.K., Zheng Z., Cheng Y. On the Optimization of Kubernetes toward the Enhancement of Cloud Computing. *Mathematics*. Multidisciplinary Digital Publishing Institute, 2024. Vol. 12, No. 16. P. 2476. DOI: 10.3390/math12162476.
16. Xavier L., Brito A., Hora A., Valente M.T. Historical and Impact Analysis of API Breaking Changes: A Large-Scale Study. *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER). 2017. P. 138–147. DOI: 10.1109/SANER.2017.7884616.
17. Dackebro E. An empirical investigation into problems caused by breaking changes in API evolution. 2019. URL: <https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-255015>.
18. Robbes R., Lungu M. A Study of Ripple Effects in Software Ecosystems: (NIER Track). *2011 33rd International Conference on Software Engineering (ICSE)*: 2011 33rd International Conference on Software Engineering (ICSE). 2011. P. 904–907. DOI: 10.1145/1985793.1985940.
19. API Failures in Openstack Cloud Environments [Электронный ресурс]. *SciSpace - Paper*. 2017. URL: <https://typeset.io/papers/api-failures-in-openstack-cloud-environments-490fssl24s> (Режим доступа: 11.07.2025).
20. Brito A., Xavier L., Hora A., Valente M.T. APIDiff: Detecting API Breaking Changes. *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*: 2018 IEEE 25th International

- Conference on Software Analysis, Evolution and Reengineering (SANER). 2018. P. 507–511. DOI: 10.1109/SANER.2018.8330249.
21. Waseem M., Liang P., Shahin M., Ahmad A., Nassab A.R. On the Nature of Issues in Five Open Source Microservices Systems: An Empirical Study. Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering. New York, NY, USA: Association for Computing Machinery, 2021. P. 201–210. DOI: 10.1145/3463274.3463337.
 22. Chen F., Zhang L., Lian X. A Systematic Gray Literature Review: The Technologies and Concerns of Microservice Application Programming Interfaces. *Software: Practice and Experience*. John Wiley & Sons, Ltd, 2021. Vol. 51, No. 7. P. 1483–1508. DOI: 10.1002/spe.2967.
 23. REST APIs don't need a versioning strategy - they need a change strategy [Электронный ресурс]. Ben Morris. *Agile enterprise architecture*. URL: <https://www.ben-morris.com/rest-apis-dont-need-a-versioning-strategy-they-need-a-change-strategy/> (Режим доступа: 29.07.2025).
 24. Lehman M.M. On Understanding Laws, Evolution, and Conservation in the Large-Program Life Cycle. *Journal of Systems and Software*. 1979. Vol. 1. P. 213–221. DOI: 10.1016/0164-1212(79)90022-0.
 25. Teivonen P., Staging Environment Implementation in a Software Delivery Automation Pipeline. Bachelor's Thesis, Oulu University of Applied Sciences, Oulu, Finland, 2024. [Электронный ресурс]. 2024. URL: <http://www.theseus.fi/handle/10024/862900> (Режим доступа: 14.07.2025).
 26. Everett G.D., Jr R.M. Software Testing: Testing Across the Entire Software Development Life Cycle. John Wiley & Sons, 2007. 279 p.
 27. Fowler S.J. Production-Ready Microservices: Building Standardized Systems Across an Engineering Organization. O'Reilly Media, Inc., 2016. 172 p.

28. Desikan S., Ramesh G. Software Testing: Principles and Practice. Pearson Education India, 2006. 508 p.
29. Beilharz J., Wiesner P., Boockmeyer A., Brokhausen F., Behnke I., Schmid R., Pirl L., Thamsen L. Towards a Staging Environment for the Internet of Things. *2021 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops): 2021 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*. 2021. P. 312–315. DOI: 10.1109/PerComWorkshops51409.2021.9431087.
30. Introduction | Pact Docs [Електронний ресурс]. 2022. URL: <https://docs.pact.io/> (Режим доступу: 14.07.2025).
31. Carver J.C., Hong N.P.C., Thiruvathukal G.K. Software Engineering for Science. CRC Press, 2016. 334 p.
32. Loechel L., Akbayin S.-R., Grünwald E., Kiesel J., Strelnikova I., Janke T., Pallas F. Hook-in Privacy Techniques for gRPC-Based Microservice Communication. Web Engineering. / ed. by Stefanidis K., Systä K., Matera M., Heil S., Kondylakis H., Quintarelli E. Cham: Springer Nature Switzerland, 2024. P. 215–229. DOI: 10.1007/978-3-031-62362-2_15.
33. Viotti J.C., Kinderkhedia M. A Survey of JSON-Compatible Binary Serialization Specifications. arXiv, 2022. DOI: 10.48550/arXiv.2201.02089.
34. Babal H. gRPC Microservices in Go. Велика Британія: Manning, 2024, 200 p.
35. Friesen J. Java XML and JSON: Document Processing for Java SE. Apress, 2019. 535 p.
36. Richardson C. Microservices Patterns: With examples in Java. Manning, 2018. 520 p.

37. Söylemez M., Tekinerdogan B., Tarhan A.K. Microservice Reference Architecture Design: A Multi-Case Study. *Software: Practice and Experience*. John Wiley & Sons, Ltd, 2024. Vol. 54, No. 1. P. 58–84. DOI: 10.1002/spe.3241.
38. Humble J., Farley D. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Pearson Education, 2010. 956 p.
39. Servile, V., Continuous Deployment. Сполучені Штати Америки: O'Reilly Media, 2024
40. Bhuyan A.P. Microservices Design Patterns: Best Practices for Cloud-Native Architectures. Aditya Pratap Bhuyan, 2024. 750 p.
41. Serbout S., Pautasso C. An Empirical Study of Web API Versioning Practices. *Web Engineering*. / ed. by Garrigós I., Murillo Rodríguez J.M., Wimmer M. Cham: Springer Nature Switzerland, 2023. P. 303–318. DOI: 10.1007/978-3-031-34444-2_22.
42. Knoche H., Hasselbring W. Continuous API Evolution in Heterogenous Enterprise Software Systems. *2021 IEEE 18th International Conference on Software Architecture (ICSA)*: 2021 IEEE 18th International Conference on Software Architecture (ICSA). 2021. P. 58–68. DOI: 10.1109/ICSA51549.2021.00014.
43. Koçi R., Franch X., Jovanovic P., Abelló A. Classification of Changes in API Evolution. *2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC)*: 2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC). 2019. P. 243–249. DOI: 10.1109/EDOC.2019.00037.
44. Ярошинський М.С., Сіроткін О.В., Сінько Д.П., Гунько С.Б., Манолук Д.О. Коректність Плaskої Класифікації. *Електронне моделювання*. 2023. Vol. 45, No. 2. P. 34–43. DOI: 10.15407/emodel.45.02.034.

45. Gos K., Zabierowski W. The Comparison of Microservice and Monolithic Architecture. *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*: 2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH). 2020. P. 150–153. DOI: 10.1109/MEMSTECH49584.2020.9109514.
46. Richards M., Ford N. Fundamentals of Software Architecture: An Engineering Approach. 1st edition. Beijing Boston Farnham Sebastopol Tokyo: O'Reilly Media, 2020. 419 p.
47. Bass L., Clements P., Kazman R. Software Architecture in Practice. 3rd edition. Upper Saddle River, NJ Munich: Addison-Wesley Professional, 2012. 624 p.
48. Woods E. Software Architecture in a Changing World. *IEEE Software*. 2016. Vol. 33. P. 94–97. DOI: 10.1109/MS.2016.149.
49. Clark T., Barn B.S. Event Driven Architecture Modelling and Simulation. *2011 IEEE 6th International Symposium on Service Oriented System (SOSE)*: Proceedings of 2011 IEEE 6th International Symposium on Service Oriented System (SOSE). 2011. P. 43–54. DOI: 10.1109/SOSE.2011.6139091.
50. Ahmed B.S., Azzalin T., Kassler A., Thore A., Lindbäck H. Smart Manufacturing: MLOps-Enabled Event-Driven Architecture for Enhanced Control in Steel Production. *Journal of Systems and Software*. 2025. Vol. 230. P. 112542. DOI: 10.1016/j.jss.2025.112542.
51. Zhang X., Liu Y., Gu S., Tian Y., Gao Y. Event-Driven Edge Agent Framework for Distributed Control in Distribution Networks. *Energies*. Multidisciplinary Digital Publishing Institute, 2025. Vol. 18, No. 11. P. 2734. DOI: 10.3390/en18112734.

52. Yıldız A., Demirörs O. MicroArc: Event Driven Analysis and Design Method for Microservices. *Procedia Computer Science*. 2025. Vol. 263. P. 583–590. DOI: 10.1016/j.procs.2025.07.070.
53. Mosser S., Chauvel F., Blay-Fornarino M., Riveill M. Web Services Composition: Mashups Driven Orchestration Definition. *2008 International Conference on Computational Intelligence for Modelling Control & Automation: 2008 International Conference on Computational Intelligence for Modelling Control & Automation*. 2008. P. 284–289. DOI: 10.1109/CIMCA.2008.96.
54. Carducci M. Orchestrated Event-Driven Abstract Style. *Mastering Software Architecture: A Comprehensive New Model and Approach*. / ed. by Carducci M. Berkeley, CA: Apress, 2025. P. 271–288. DOI: 10.1007/979-8-8688-0410-6_18.
55. Abdelmoumen A., Benzadri Z., Bouassida Rodriguez I. A Service-Oriented Framework for Resource-Aware System-of-Systems Modeling in IoT Environments. *Service-Oriented Computing – ICSOC 2024 Workshops*. / ed. by Kallel S., Raibulet C., Bouassida Rodriguez I., Faci N., Bennaceur A., Cheikhrouhou S., Ben Ayed L., Sellami M., Nakagawa E.Y., Ben Halima R. Singapore: Springer Nature, 2026. P. 163–169. DOI: 10.1007/978-981-96-7423-7_14.
56. Zahiri N., Babamir S.M. A Pattern-Based and Multi-Objective Architecture for Selecting Optimal Compositions of Web Services. *Expert Systems with Applications*. 2026. Vol. 296. P. 128674. DOI: 10.1016/j.eswa.2025.128674.
57. Hannousse A., Yahiouche S. Securing Microservices and Microservice Architectures: A Systematic Mapping Study. *Computer Science Review*. 2021. Vol. 41. P. 100415. DOI: 10.1016/j.cosrev.2021.100415.

58. Faustino D., Gonçalves N., Portela M., Rito Silva A. Stepwise Migration of a Monolith to a Microservice Architecture: Performance and Migration Effort Evaluation. *Performance Evaluation*. 2024. Vol. 164. P. 102411. DOI: 10.1016/j.peva.2024.102411.
59. Milanović D.M. Airbnb Microservice Architecture. *Medium*. 2024. URL: <https://medium.com/@techworldwithmilan/airbnb-microservice-architecture-bd1986c73719>.
60. Gluck A. Introducing Domain-Oriented Microservice Architecture [Электронный ресурс]. *Uber Blog*. 2020. URL: <https://www.uber.com/en-IN/blog/microservice-architecture/> (Режим доступа: 30.07.2025).
61. Blog N.T. Rebuilding Netflix Video Processing Pipeline with Microservices [Электронный ресурс]. *Medium*. 2024. URL: <https://netflixtechblog.com/rebuilding-netflix-video-processing-pipeline-with-microservices-4e5e6310e359> (Режим доступа: 30.07.2025).
62. What Is MSE? - Microservices Engine - Alibaba Cloud Documentation Center [Электронный ресурс]. URL: <https://www.alibabacloud.com/help/en/mse/product-overview/what-is-mse> (Режим доступа: 17.08.2025).
63. Microservices Architecture on Google App Engine | App Engine standard environment for Python 2 [Электронный ресурс]. *Google Cloud*. URL: <https://cloud.google.com/appengine/docs/legacy/standard/python/microservices-on-app-engine> (Режим доступа: 17.08.2024).
64. Cloud Native Computing Foundation [Электронный ресурс]. *CNCF*. URL: <https://www.cncf.io/> (Режим доступа: 03.07.2025).
65. Jaegertracing/Jaeger. Jaeger - Distributed Tracing Platform, 2024. URL: <https://github.com/jaegertracing/jaeger>.

66. Shkuro Y. Mastering Distributed Tracing: Analyzing performance in microservices and complex systems. 1st edition. Packt Publishing, 2019. 446 p.
67. Janovic J. Cisco ACI: Zero to Hero: A Comprehensive Guide to Cisco ACI Design, Implementation, Operation, and Troubleshooting. Berkeley, CA: Apress, 2023. DOI: 10.1007/978-1-4842-8838-2.
68. Santos Filho A., Rodríguez R.J., Feitosa E.L. Automated broken object-level authorization attack detection in REST APIs through OpenAPI to colored petri nets transformation. *International Journal of Information Security*. 2025. Vol. 24, No. 2. P. 83. DOI: 10.1007/s10207-024-00970-5.
69. Attouche L., Baazizi M.-A., Colazzo D., Ghelli G., Imo K.C., Klessinger S., Sartiani C., Scherzinger S. JTutor: JSON Schema Validation Explained. Proceedings of the 19th International Symposium on Database Programming Languages. New York, NY, USA: Association for Computing Machinery, 2025. P. 1–6. DOI: 10.1145/3735106.3736532.
70. Sharma A., Kamthania D. QUIC Protocol Based Monitoring Probes for Network Devices Monitor and Alerts. *Smart Sensor Networks: Analytics, Sharing and Control*. / ed. by Singh U., Abraham A., Kaklauskas A., Hong T.-P. Cham: Springer International Publishing, 2022. P. 127–150. DOI: 10.1007/978-3-030-77214-7_6.
71. Teodor M., Mocanu B.-C., Negru C., Pop F. QUICPot. A HTTP/3 Protocol Honeypot. Innovative Security Solutions for Information Technology and Communications. / ed. by Morogan L., Roenne P., Bica I. Cham: Springer Nature Switzerland, 2025. P. 207–222. DOI: 10.1007/978-3-031-87760-5_15.
72. Koutras D., Dimitrakopoulos G., Malamas V., Kotzanikolaou P., Douligeris C. Comparative Analysis and Implementation of HTTP3, MQTT, and CoAP for IoT Applications. Proceedings of the 28th Pan-Hellenic Conference on

- Progress in Computing and Informatics. New York, NY, USA: Association for Computing Machinery, 2025. P. 127–132. DOI: 10.1145/3716554.3716830.
73. Bernhardt M. Reactive Web Applications: Covers Play, Akka, and Reactive Streams. Shelter Island, New York: Manning Publications Co, 2016. 302 p.
 74. Lu J., Mo C., Cao P., Li G. Design and Implementation of a Real-Time Collaborative Multi-End Interaction Platform Based on WebSocket and WebRTC. *2025 4th International Symposium on Computer Applications and Information Technology (ISCAIT)*: 2025 4th International Symposium on Computer Applications and Information Technology (ISCAIT). 2025. P. 2181–2186. DOI: 10.1109/ISCAIT64916.2025.11010474.
 75. George N., Thomas K., A. M.Y., Joy J., Joby J.E. Streamlining Communication and Collaboration with Nexus Chat. *AIP Conference Proceedings*. 2025. Vol. 3260, No. 1. P. 020048. DOI: 10.1063/5.0259589.
 76. Deshpande K., Jain A., Abhinav, Mishra S., Goel S., Garg R. Empowering Real-Time Communication: A Seamless Chatting System Using Websocket. *Innovative Computing and Communications*. / ed. by Hassanien A.E., Anand S., Jaiswal A., Kumar P. Singapore: Springer Nature, 2025. P. 405–417. DOI: 10.1007/978-981-97-4152-6_29.
 77. Aswad I., Bustamin A., Arisandy Safruddin R., Niswar M. Performance Evaluation of Microservices Communication with REST, GraphQL, and gRPC. *International Journal of Electronics and Telecommunications*; 2024; vol. 70; No 2; 429-436. Polish Academy of Sciences Committee of Electronics and Telecommunications, 2024. URL: <https://journals.pan.pl/dlibra/publication/149562/edition/131803>.
 78. Raymond E. The Art of UNIX Programming. 1st edition. Boston: Addison-Wesley, 2003. 560 p.

79. Hindman B., Konwinski A., Zaharia M., Ghodsi A., Joseph A.D., Katz R., Shenker S., Stoica I. Mesos: A Platform for {Fine-Grained} Resource Sharing in the Data Center. *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*. 2011. URL: <https://www.usenix.org/conference/nsdi11/mesos-platform-fine-grained-resource-sharing-data-center>.
80. Bakhshi Zadi Mahmoodi A., Peltonen E. Kubernetes Task-Offloading Framework for Edge-Cloud Continuum in Vehicular Computing. Proceedings of the 14th International Conference on the Internet of Things. New York, NY, USA: Association for Computing Machinery, 2025. P. 146–149. DOI: 10.1145/3703790.3703807.
81. Jain D., Blossom J., Hayes J., Gibson H., Rifas-Shimann S., Gold D.R. RINX 2.0: A Containerized Climate Raster Information Extraction System on OpenShift Cloud Environment. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*. Copernicus GmbH, 2025. Vol. X-G-2025. P. 391–395. DOI: 10.5194/isprs-annals-X-G-2025-391-2025.
82. Docker overview [Електронний ресурс]. *Docker Documentation*. 100 AD. URL: <https://docs.docker.com/get-started/docker-overview/> (Режим доступу: 21.07.2025).
83. Swarm mode [Електронний ресурс]. *Docker Documentation*. 800. URL: <https://docs.docker.com/engine/swarm/> (Режим доступу: 22.07.2025).
84. Schenker G.N., Saito H., Lee H.C.C., Hsu K.J.C. Getting Started with Containerization: Reduce the Operational Burden on Your System by Automating and Managing Your Containers. Packt Publishing, 2019. URL: <https://books.google.com.ua/books?id=ZDqPDwAAQBAJ>.

85. Що таке Kubernetes? [Електронний ресурс]. URL: <https://kubernetes.io/uk/docs/concepts/overview/what-is-kubernetes/> (дата звернення: 21.08.2024).
86. Alremeithi K., Sealy W. The Use of Digital Twin for Mobile Robot Swarm Task Allocation. *Manufacturing Letters*. 2024. Vol. 41. P. 1200–1208. DOI: 10.1016/j.mfglet.2024.10.001.
87. Fathoni H., Yang C.-T., Huang C.-Y., Chen C.-Y. Empowered edge intelligent aquaculture with lightweight Kubernetes and GPU-embedded. *Wireless Networks*. 2024. Vol. 30, No. 9. P. 7321–7333. DOI: 10.1007/s11276-023-03592-2.
88. K9s - Manage Your Kubernetes Clusters In Style [Електронний ресурс]. URL: <https://k9scli.io/> (Режим доступу: 06.07.2025).
89. Kurrewar S., Dhokane S., Dahake A., Yadav R.K., Wyawahare N., Morris N.C. Streamlining Kubernetes Deployments through GitOps Methodologies. 2025 *IEEE International Students' Conference on Electrical, Electronics and Computer Science (SCEECS)*: 2025 IEEE International Students' Conference on Electrical, Electronics and Computer Science (SCEECS). 2025. P. 1–7. DOI: 10.1109/SCEECS64059.2025.10941164.
90. Wilken T., Eulisse G. Managing software build infrastructure at ALICE using Hashicorp Nomad. *EPJ Web of Conferences*. EDP Sciences, 2024. Vol. 295. P. 05013. DOI: 10.1051/epjconf/202429505013.
91. Apache Mesos [Електронний ресурс]. *Apache Mesos*. URL: <https://mesos.apache.org/> (Режим доступу: 22.07.2025).
92. D2iq-Archive/Marathon. D2iQ Archive - A Nutanix technology, 2025. URL: <https://github.com/d2iq-archive/marathon>.
93. The Definitive Platform for Modern Apps [Електронний ресурс]. *DC/OS*. URL: <https://dcos.io/> (Режим доступу: 22.07.2025).

94. Marathon: A Container Orchestration Platform for Mesos [Электронный ресурс]. URL: https://aventer-ug.github.io/marathon/index.html?utm_source=chatgpt.com (Режим доступа: 22.08.2025).
95. Sabharwal N., Pandey S., Pandey P. Infrastructure-as-Code Automation Using Terraform, Packer, Vault, Nomad and Consul: Hands-on Deployment, Configuration, and Best Practices. Berkeley, CA: Apress, 2021. DOI: 10.1007/978-1-4842-7129-2.
96. Networking | Nomad | HashiCorp Developer [Электронный ресурс]. *Networking | Nomad | HashiCorp Developer*. URL: <https://developer.hashicorp.com/nomad/docs/networking> (Режим доступа: 22.08.2025).
97. Sobhani G., Haque I., Sharma T. It Works (Only) on My Machine: A Study on Reproducibility Smells in Ansible Scripts. *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*: 2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR). 2025. P. 384–395. DOI: 10.1109/MSR66628.2025.00069.
98. Nasiri R., Kumara I., Tamburri D.A., van den Heuvel W.-J. Towards a Taxonomy of Infrastructure as Code Misconfigurations: An Ansible Study. *Service-Oriented Computing*. / ed. by Aiello M., Barzen J., Dustdar S., Leymann F. Cham: Springer Nature Switzerland, 2025. P. 83–103. DOI: 10.1007/978-3-031-72578-4_5.
99. Friyanto A. NETCONF, RESTCONF and ANSIBLE: Comparison of Network Device Configuration Method Transformations. *AIP Conference Proceedings*. 2025. Vol. 3200, No. 1. P. 040016. DOI: 10.1063/5.0255250.
100. Smith S.R., Membrey P. Beginning Ansible Concepts and Application: Provisioning, Configuring, and Managing Servers, Applications, and Their

- Dependencies. Berkeley, CA: Apress, 2022. DOI: 10.1007/978-1-4842-8173-4.
101. Perumal A.P., Perumal A.P. Building Resilient Systems: The Role of Containers and Kubernetes, IGI Global Scientific Publishing, 1 AD. DOI: 10.4018/979-8-3373-0365-9.ch013.
 102. Barr A.B., Lavi O., Naor Y., Rampal S., Tavori J. Performance Comparison of Service Mesh Frameworks: The MTLS Test Case. *NOMS 2025-2025 IEEE Network Operations and Management Symposium*: NOMS 2025-2025 IEEE Network Operations and Management Symposium. 2025. P. 1–6. DOI: 10.1109/NOMS57970.2025.11073712.
 103. Zhang X., Zhao P., Jaskolka J. Navigating the DevOps Landscape. *Journal of Systems and Software*. 2025. Vol. 223. P. 112331. DOI: 10.1016/j.jss.2024.112331.
 104. Vitumbiko M., Kim Y. Design of Network Setup Automation Using Gitops Operation. *2025 International Conference on Information Networking (ICOIN)*: 2025 International Conference on Information Networking (ICOIN). 2025. P. 402–405. DOI: 10.1109/ICOIN63865.2025.10993110.
 105. Jouda B., Owies N., Atoum I. Software Configuration Management in Evolutionary Processes: Tools and Best Practices. *2025 16th International Conference on Information and Communication Systems (ICICS)*: 2025 16th International Conference on Information and Communication Systems (ICICS). 2025. P. 1–6. DOI: 10.1109/ICICS65354.2025.11073118.
 106. Manuel S. Nilslice/Protolock. 2025. URL: <https://github.com/nilslice/protolock>.
 107. Habib A., Shinnar A., Hirzel M., Pradel M. Finding Data Compatibility Bugs with JSON Subschema Checking. *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY,

- USA: Association for Computing Machinery, 2021. P. 620–632. DOI: 10.1145/3460319.3464796.
108. Lam P., Dietrich J., Pearce D.J. Putting the Semantics into Semantic Versioning. Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. New York, NY, USA: Association for Computing Machinery, 2020. P. 157–179. DOI: 10.1145/3426428.3426922.
109. Regulation - 2024/2847 - EN - EUR-Lex [Электронный ресурс]. URL: <https://eur-lex.europa.eu/eli/reg/2024/2847/oj/eng> (Режим доступа: 06.07.2025).
110. Software Considerations in Airborne Systems and Equipment Certification (*Only for Aviation-Related Projects*) | Standards [Электронный ресурс]. URL: <https://standards.nasa.gov/standard/NASA/RTCA-DO-178> (Режим доступа: 06.07.2025).
111. Sun R., Zhong D., Li W., Lu M., Ding Y., Xu Z., Gong H., Zha Y. A Safety Analysis Method of Airborne Software Based on ARP4761. *Journal of Physics: Conference Series*. IOP Publishing, 2020. Vol. 1673, No. 1. P. 012045. DOI: 10.1088/1742-6596/1673/1/012045.
112. Weedy B.M., Cory B.J., Jenkins N., Ekanayake J.B., Strbac G. Electric Power Systems. John Wiley & Sons, 2012. 498 p.
113. Angelakos J. PostgreSQL Mistakes and How to Avoid Them. Manning, 2025. URL: <https://books.google.com.ua/books?id=X4dlEQAAQBAJ>.

ДОДАТОК А

Список публікацій здобувача

1. М.С. Ярошинський, О.В. Сіроткін, Д.П. Сінько, С.Б. Гунько, Д.О. Манолук, ‘Коректність пласкої класифікації’, Електронне моделювання Т. 45, № 2 (2023) с. 34-43 doi: /10.15407/emodel.45.02.034. Фахове видання категорії Б. (Особистий внесок – Брав участь у аналізі матеріалів, розробці та формальному обґрунтуванні математичного апарату для оцінки коректності пласких класифікацій. На основі введених операцій та поняття відносної відстані між класами, запропонував формальну міру відмінності між двома класами).
2. А.М.Примушко, І.В. Пучко, М.С. Ярошинський, Д.П. Сінько, ‘Програмний дизайн розподіленої високонавантаженої системи електроенергетичної мережі на базі моделі акторів із застосуванням смарт-контрактів’, Електронне моделювання Т. 46, № 3 (2024) с. 57-72 doi: /10.15407/emodel.46.03. Фахове видання категорії Б. (Особистий внесок – технічне обґрунтуванні архітектурних рішень, що реалізують запропоновану високорівневу концепцію. Визначенні структури даних, що передаються між вузлами системи, запропонував та формалізував JSON-формат із обов'язковим набором параметрів, запропонував модель поведінки IoT-пристроїв у системі, визначивши три ключові стани: Активний, Пасивний та Неактивний).
4. A. Prymushko, I. Puchko, M. Yaroshynskyi, D. Sinko, H. Kravtsov, and V. Artemchuk, ‘Efficient State Synchronization in Distributed Electrical Grid Systems Using Conflict-Free Replicated Data Types’, IoT, vol. 6, no. 1, p. 6, Jan. 2025, doi: 10.3390/iot6010006. Indexed in Scopus Q1. (Особистий внесок – брав участь у плануванні і проведенні експериментів, брав участь у візуалізації даних, брав участь у формалізації структури стану вузла та

потоків комунікації всередині мережі, формуванні і моделюванні графу роботи мережі).

5. O. Sirotkin, A. Prymushko, I. Puchko, H. Kravtsov, M. Yaroshynskyi, and V. Artemchuk, ‘Parallel Simulation Using Reactive Streams: Graph-Based Approach for Dynamic Modeling and Optimization’, *Computation*, vol. 13, no. 5, p. 103, Apr. 2025, doi: 10.3390/computation13050103. Indexed in Scopus Q2. (Особистий внесок – брав участь у аналізі та інтерпретації результатів, в розробці математичної моделі для динамічного моделювання).
6. M. Yaroshynskyi, A. Prymushko, I. Puchko, O. Sirotkin, and D. Sinko, ‘Akka as a tool for modelling and managing a smart grid system’, *J. Edge Comp.*, vol. 4, no. 1, pp. 105–115, May 2025, doi: 10.55056/jec.822. Indexed in Scopus. (Особистий внесок – брав участь у аналізі ключових викликів в управлінні інтелектуальними мережами та в розробці моделі управління інтелектуальною мережею на основі ієрархічної структури акторів Akka, моделював архітектуру мережі — від регіонів до окремих пристроїв, брав участь у концептуалізації та впровадженні підходу, що поєднує моделювання та реальне управління в єдиній акторній архітектурі).
7. M. Yaroshynskyi, I. Puchko, A. Prymushko, H. Kravtsov, and V. Artemchuk, ‘Investigating the Evolution of Resilient Microservice Architectures: A Compatibility-Driven Version Orchestration Approach’, *Digital*, vol. 5, no. 3, p. 27, July 2025, doi: 10.3390/digital5030027. Indexed in Scopus Q2. (Особистий внесок – брав участь в аналізі існуючих стратегій управління еволюцією API, запропонував підхід з Compatibility-Driven Version Orchestrator, створення математичної моделі поширення збоїв через несумісні зміни API, брав участь в організації експериментів, проведення експериментів, візуалізація результатів).

8. О.В. Сіроткін, М.С. Ярошинський, Д.П. Сінько, С.Б. Гунько, Д.О. Манолук, ‘Моделювання у фазовому просторі під-станів’, Електронне моделювання Т. 47, № 3 (2025), с. 28-45 doi: /10.15407/emodel.47.03.028 Фахове видання категорії Б. (Особистий внесок – брав участь у побудові концептуальної моделі, симуляції моделі)
9. Ярошинський М.С., Пучко І.В. Способи розв’язання проблеми асинхронності зміни прикладного програмного інтерфейса в мікросервісній архітектурі. Електронне моделювання Т. 47, № 4 (2025) с. 57-72 doi: /10.15407/emodel.47.04.057. Фахове видання категорії Б. (Особистий внесок – брав участь в аналізі існуючих стратегій управління еволюцією API, брав участь у створення математичної моделі поширення збоїв через несумісні зміни API).
10. М.С. Ярошинський, ‘Захист від атак підробки та перехоплення за допомогою jwt для неконфіденційної інформації’, Матеріали науково-практичної конференції ‘XLI науково-технічна конференція молодих вчених та спеціалістів інституту проблем моделювання в енергетиці ім. Г.Є. Пухова НАН України’, с. 141-145, Україна, 2023, URL: <https://ipme.kiev.ua/konferencii/konferenciya-molodix-vchenix-2023>
11. І.В. Пучко, А.М. Примушко, М.С. Ярошинський, Г.О. Кравцов, ‘Підвищення резильєнтності динамічних систем при синхронізації станів за допомогою CRDT’, Матеріали науково-практичної конференції ‘Резильєнтність динамічних систем, с. 50–52, Київ, Україна, 2024 URL: <https://ipme.kiev.ua/konferencii/naukovo-praktichna-konferenciya-rds-2024>
12. Prymushko A., Yaroshynskyi M., Puchko I. Representation and synchronization of states of distributed electrical grid systems based on conflict free replicated data types. 2024 14th International Conference on Dependable Systems,

Services and Technologies (DESSERT), Athens, Greece, 2024, pp. 1-5, doi:
/10.1109/DESSERT65323.2024.11122143. Indexed in Scopus.

ДОДАТОК Б

Модуль ServicesDAO.scala*

```
import anorm.*
import anorm.SqlParser.*
import anorm.postgresql.*
import com.google.inject.{Inject, Singleton}
import controllers.{ApiVersion, ServiceVersion}
import play.api.db.Database
import play.api.libs.json.JsValue

import scala.concurrent.ExecutionContext

@Singleton
class ServicesDAO @Inject(db: Database)(implicit dispatcher:
ExecutionContext):

  private val request =
    """WITH api_version_groups AS (
      | SELECT
      |   api_id,
      |   api_type,
      |   api_version_major,
      |   api_version_minor,
      |   COUNT(DISTINCT service_id) as service_count
      | FROM service_api_versions
      | GROUP BY api_id, api_type, api_version_major, api_version_minor
      | ),
      | most_common_api_versions AS (
      | SELECT
      |   api_id,
      |   api_type,
      |   api_version_major,
      |   api_version_minor
      | FROM (
```

```

| SELECT
|   *,
|   ROW_NUMBER() OVER (
|     PARTITION BY api_id, api_type
|     ORDER BY service_count DESC, api_version_major DESC,
api_version_minor DESC
|   ) as rn
| FROM api_version_groups
| ) ranked
| WHERE rn = 1
| ),
| service_data AS (
| SELECT
|   s.service_id,
|   s.service_name,
|   s.service_version_major,
|   s.service_version_minor,
|   json_agg(
|     json_build_object(
|       'api_id', s.api_id,
|       'api_type', s.api_type,
|       'api_version_major', m.api_version_major,
|       'api_version_minor', m.api_version_minor
|     )
|   ) as apis
| FROM service_api_versions s
|   JOIN most_common_api_versions m ON s.api_id = m.api_id AND
s.api_type = m.api_type
|   GROUP BY s.service_id, s.service_name, s.service_version_major,
s.service_version_minor
| ),
| latest_services AS (
| SELECT *
| FROM (

```

```

| SELECT *,
|   ROW_NUMBER() OVER (
|     PARTITION BY service_id
|     ORDER BY service_version_major DESC, service_version_minor DESC
|   ) as rn
| FROM service_data
| ) ranked_services
| WHERE rn = 1
|)
|SELECT
| service_id,
| service_name,
| service_version_major,
| service_version_minor,
| apis
|FROM latest_services
|ORDER BY service_name;
|"".stripMargin

```

```

private val serviceVersionParser: RowParser[ServiceVersion] = for {
  service_id      <- str("service_id")
  service_name    <- str("service_name")
  apis            <- get[JsValue]("apis")
  service_version_major <- int("service_version_major")
  service_version_minor <- int("service_version_minor")
} yield ServiceVersion(
  service_id = service_id,
  service_name = service_name,
  api = apis.validate[Seq[ApiVersion]].getOrElse(Seq.empty[ApiVersion]),
  service_version_major = service_version_major,
  service_version_minor = service_version_minor
)

```

```

def insert(sv: ServiceVersion): Seq[Int] = db.withConnection { implicit conn =>

```



```

sv.api.flatMap { apiVersion =>
  val batch = Seq[NamedParameter](
    "service_id" -> sv.service_id,
    "service_name" -> sv.service_name,
    "api_id" -> apiVersion.api_id,
    "api_type" -> apiVersion.api_type,
    "api_version_major" -> apiVersion.api_version_major,
    "api_version_minor" -> apiVersion.api_version_minor,
    "service_version_major" -> sv.service_version_major,
    "service_version_minor" -> sv.service_version_minor
  )
  BatchSql(
    """
    INSERT INTO service_api_versions (
      service_id, service_name,
      api_id, api_type, api_version_major, api_version_minor,
      service_version_major, service_version_minor
    ) VALUES (
      {service_id}, {service_name},
      {api_id}, {api_type}, {api_version_major}, {api_version_minor},
      {service_version_major}, {service_version_minor}
    )
    """,
    batch
  ).execute()
}

def getRequiredVersions: Seq[ServiceVersion] = db.withConnection { implicit x
=>
  SQL(request).as(serviceVersionParser.*)
}

```

* Джерело: побудовано автором

ДОДАТОК В

Розміщені тестові сервіси на системі Kubernetes

```
Context: microk8s [RW]
Cluster: microk8s-cluster
User: admin
K9s Rev: v0.50.12 ⚡ v0.50.13
K8s Rev: v1.31.13
CPU: 7%
MEM: 21%
```

<0> all <a> ...
<1> default <ctrl-d> ...
<d> ...
<e> ...
<?> ...
<shift-j> ...

pods(default)[14]

NAME	PF	READY	STATUS	RESTARTS	CPU	%CPU/R	CPU/L
egydata-play-depl-69968f975d-cgtdl	●	1/1	Running	11	9	n/a	n/a
grafana-85d8cfd9b9-6mfzg	●	1/1	Running	0	7	n/a	n/a
prometheus-alertmanager-0	●	1/1	Running	0	2	n/a	n/a
prometheus-kube-state-metrics-7fb455bd77-4246f	●	1/1	Running	2	5	n/a	n/a
prometheus-prometheus-node-exporter-pnpc9	●	1/1	Running	1	17	n/a	n/a
prometheus-prometheus-node-exporter-pz8vf	●	1/1	Running	0	12	n/a	n/a
prometheus-prometheus-pushgateway-85f98dc7b7-lr5dh	●	1/1	Running	0	2	n/a	n/a
prometheus-server-f9764598-q7cnk	●	2/2	Running	0	57	n/a	n/a
svc1-blue-5b58945dcc-fz445	●	1/1	Running	0	3	n/a	n/a
svc2-blue-79fb55f77-c7r6s	●	1/1	Running	0	2	n/a	n/a
svc3-blue-67f8d5f957-gdqs8	●	1/1	Running	0	0	n/a	n/a
test-backend-deployment-b4885b96c-kb74n	●	1/1	Running	167	2	n/a	n/a
test-backend-deployment-b4885b96c-lbj98	●	1/1	Running	225	3	n/a	n/a
test-backend-deployment-b4885b96c-tqgmX	●	1/1	Running	226	2	n/a	n/a

<pod>

Рисунок Б.1 Розміщені сервіси на системі Kubernetes. Інтерфейс
аплікації k9s

Джерело: побудовано автором

ДОДАТОК Г

Сценарій розміщення Helm.

Частина deployment.yaml*

```
{{- range .Values.microservices }}
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .name }}-{{ $.Values.color }}
  labels:
    app: {{ .name }}
    color: {{ $.Values.color }}
spec:
  replicas: 1
  selector:
    matchLabels:
      app: {{ .name }}
      color: {{ $.Values.color }}
  template:
    metadata:
      labels:
        app: {{ .name }}
        color: {{ $.Values.color }}
    spec:
      containers:
        - name: {{ .name }}
          image: {{ .image }}:{{ .tag }}
          ports:
            - containerPort: {{ .port }}
          env:
            - name: COLOR
              value: "{{ $.Values.color }}"
            {{- if eq .name "svc1" }}
            - name: GRPC_SVC2_ADDRESS
```

```

        value: "svc2-{{ $.Values.color }}"
      {{- end }}
    {{- if eq .name "svc2" }}
    - name: REMOTEAPI_CLIENT_URL
      value: "http://svc3-{{ $.Values.color }}:8080"
    {{- end }}
  ---
{{- end }}

```

services.yaml*

```

{{- range .Values.microservices }}
apiVersion: v1
kind: Service
metadata:
  name: {{ .name }}-{{ $.Values.color }}
  labels:
    app: {{ .name }}
    color: {{ $.Values.color }}
  annotations:
    {{- if eq .name "svc1" }}
    prometheus.io/scrape: "true"
    prometheus.io/port: "8080"
    prometheus.io/path: "/actuator/prometheus"
    meta.helm.sh/release-name: {{ $.Release.Name }}
    meta.helm.sh/release-namespace: {{ $.Release.Namespace }}
    {{- end }}
spec:
  selector:
    app: {{ .name }}
    color: {{ $.Values.color }}
  ports:
    - protocol: TCP
      port: 8080
      targetPort: {{ .port }}

```

```
---  
{{- end }}
```

* Джерело: побудовано автором

values.yaml**

```
- name: svc1  
  image: nyarosh/initial-service  
  tag: v0.2  
  port: 8080  
- name: svc2  
  image: nyarosh/middle-service  
  tag: v0.2  
  port: 8080  
- name: svc3  
  image: nyarosh/final-service  
  tag: v0.2  
  port: 8080
```

** Джерело: побудовано аплікацією