МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЇВСЬКИЙ АВІАЦІЙНИЙ ІНСТИТУТ»

Кваліфікаційна наукова
праця на правах рукопису

ЧИЖОВ ОЛЕКСАНДР ВІКТОРОВИЧ

УДК 004.75:004.77

ДИСЕРТАЦІЯ

ІНФОРМАЦІЙНА ТЕХНОЛОГІЯ ОПТИМІЗАЦІЇ ВИКОРИСТАННЯ ОБЧИСЛЮВАЛЬНИХ РЕСУРСІВ У ХМАРНИХ КЛАСТЕРАХ ДЛЯ ДИНАМІЧНОГО ХОСТИНГУ ВЕБ-СЕРВІСІВ

122 – Комп’ютерні науки
12 – Інформаційні технології

Подається на здобуття наукового ступеня доктора філософії.

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

_____ О.В. Чижов

Науковий керівник:
Фесенко Андрій Олексійович
кандидат технічних наук, доцент

Київ – 2026

АНОТАЦІЯ

Чижов О. В. Інформаційна технологія оптимізації використання обчислювальних ресурсів у хмарних кластерах для динамічного хостингу веб-сервісів. – Кваліфікаційна наукова праця на правах рукопису.

Дисертація на здобуття наукового ступеня доктора філософії за спеціальністю 122 «Комп'ютерні науки» (12 – Інформаційні технології). – Національний університет «Київський авіаційний інститут». Київ, 2026.

Дисертаційну роботу присвячено розробленню та обґрунтуванню інформаційної технології оптимізації використання обчислювальних ресурсів у хмарних кластерах для динамічного хостингу веб-сервісів. Розглянуто завдання масового обслуговування великої кількості слабо навантажених доменів за стохастичного характеру трафіку, нерівномірного розподілу попиту між доменами, домінування шифрованих з'єднань і жорстких вимог до доступності та швидкодії, що визначаються угодами про рівень надання послуг (SLA). Показано, що для цього сегмента характерна суперечність між прагненням до максимальної консолідації обчислювальних ресурсів і необхідністю забезпечувати передбачувану якість обслуговування, зокрема в умовах загроз доступності на прикладному рівні.

При цьому особливість досліджуваного класу систем полягає в тому, що традиційні підходи до резервування обчислювальної потужності не забезпечують належного балансу між економічною ефективністю та якістю обслуговування. Статичне закріплення ресурсу за окремими вузлами або групами сайтів призводить до фрагментації резерву та простоювання значної частини обчислювальної потужності, тоді як надмірне ущільнення розміщення доменів підвищує ризик взаємного впливу орендарів і порушення SLA. У зв'язку з цим у роботі розглянуто не лише задачу масштабування як реакцію на перевантаження, а комплексну задачу формалізованого визначення мінімально достатньої ємності кластера, стабілізації керувального контуру та підтримання працездатності системи за змінного складу доменів, сертифікатів, а також за змінної інтенсивності мережевих запитів..

Об’єкт дослідження — процеси планування, динамічного розподілу та адаптивного керування обчислювальними ресурсами в хмарних системах масового обслуговування веб-сервісів.

Предметом дослідження є математичні моделі, методи та алгоритми оптимізації використання обчислювальних ресурсів у хмарному кластері динамічного вебхостингу з урахуванням імовірнісних вимог до якості обслуговування та асинхронної організації контуру прикладного захисту.

Метою роботи: розробити інформаційну технологію оптимізації використання обчислювальних ресурсів у хмарних кластерах для динамічного хостингу веб-сервісів, яка забезпечує дотримання вимог SLA в умовах стохастичного багатодоменного навантаження та створює можливості для застосування прикладних механізмів захисту без погіршення швидкодії основного тракту оброблення легітимних запитів.

У роботі розроблено математичну модель адаптивного керування ємністю хмарного кластера масового вебхостингу, у якій інфраструктуру подано як єдиний пул обчислювальних ресурсів, а оперативну пам’ять визначено як критично обмежувальний ресурс, що задає допустиму конкурентність оброблення запитів. Для формалізації вимог до рівня обслуговування використано втратну модель Ерланга, що дало змогу пов’язати інтенсивність стохастичного багатодоменного трафіку, параметри обслуговування та мінімально достатню ємність кластера. Відмінність запропонованої моделі полягає в тому, що керування ємністю розглядається не як евристичне реагування на перевищення апаратних метрик, а як розрахункова процедура, пов’язана з імовірнісною оцінкою відмови в обслуговуванні. Це дає змогу переходити від локального аналізу завантаженості окремого сервера до оцінювання стану всього кластера як спільного ресурсу. У межах моделі враховано взаємозв’язок між кількістю одночасно оброблюваних запитів, середнім обсягом оперативної пам’яті, що припадає на один запит, доступною пам’яттю обчислювальних вузлів і допустимою ймовірністю блокування. Такий підхід забезпечує формалізовану основу для прийняття рішень щодо додавання або виведення вузлів із кластера без необґрунтованого збільшення резерву.

Удосконалено метод визначення необхідної конфігурації хмарного кластера та керування його масштабуванням на основі чисельно стійкого рекурентного обчислення ймовірності блокування, ітеративного добору мінімально достатньої ємності та механізмів порогового і часового гістерезису, що зменшують ризик автоколивань у керувальному контурі за короткочасних стохастичних флуктуацій навантаження. Запропонований метод дає змогу відокремити короткочасні випадкові сплески трафіку від стійкої зміни навантаження, що потребує зміни конфігурації кластера. Використання порогового гістерезису запобігає надмірно частому перемиканню станів у разі незначних коливань розрахункової ємності, а часовий гістерезис забезпечує підтвердження тенденції перед виконанням масштабування. Завдяки цьому керувальний контур зберігає стабільність у сценаріях, характерних для масового вебхостингу, де навантаження окремих доменів може змінюватися нерівномірно, імпульсно та несинхронно. Практичним наслідком є зменшення кількості необґрунтованих операцій масштабування та підтримання стабільності обслуговування без надмірного резервування ресурсу.

Розроблено метод асинхронного контуру прикладного захисту, який формалізує функціонування підсистеми відкладеного аналізу HTTP/HTTPS-запитів поза критичним шляхом обслуговування, описує динаміку багатоканальної системи з обмеженою чергою, механізм відкладеної верифікації, своєчасне виявлення ознак атаки та умови реакції на прикладні атаки. Це дало змогу обґрунтувати підхід, за якого ресурсоємний аналіз трафіку не погіршує швидкодії оброблення легітимного трафіку.

Набув подальшого розвитку метод оцінювання ресурсної ефективності консолідації в масовому хмарному вебхостингу, яка базується на інтегральному порівнянні традиційної архітектури спільного хостингу та динамічного хмарного кластера за обсягом задіяного ресурсу в часі та коефіцієнтом ефективності консолідації.

На основі одержаних результатів обґрунтовано архітектуру динамічного хмарного кластера, у якій обчислювальні вузли функціонують без збереження стану, а зберігання даних винесено до окремого шару. У межах цієї архітектури передбачено централізовану

керовану точку входу для термінування TLS-з’єднань, збирання експлуатаційних показників, керованої маршрутизації запитів і реалізації керувальних впливів під час масштабування. Сформовано дослідницький програмний прототип, у межах якого реалізовано основні компоненти запропонованої інформаційної технології, необхідні для натурної експериментальної перевірки розроблених моделей і архітектурних рішень. Експериментальна частина роботи спрямована на перевірку працездатності запропонованих рішень у сценаріях, наближених до умов експлуатації багатодоменного хостингового середовища. Досліджено масштабованість центральної точки входу під час оброблення HTTPS-запитів, роботу модуля розрахунку ємності та масштабування, ресурсну ефективність адаптивного кластерного підходу порівняно з традиційною моделлю спільного хостингу, а також стійкість системи до HTTP flood-навантаження за наявності асинхронного контуру прикладного захисту. Отримані результати підтверджують можливість застосування запропонованої інформаційної технології для підвищення щільності консолідації ресурсів, зменшення непродуктивного резервування та підтримання вимог SLA у хмарному кластері динамічного вебхостингу.

Практичне значення одержаних результатів полягає в можливості їх використання під час проєктування та модернізації інфраструктури хмарного вебхостингу, зокрема для побудови систем автоматичного масштабування, визначення мінімально достатньої конфігурації кластера, оцінювання ефективності консолідації ресурсів і впровадження прикладного захисту без включення ресурсоємного аналізу до критичного шляху оброблення запитів. Запропоновані моделі та методи можуть бути використані хостинговими провайдерами, розробниками багатодоменних вебплатформ і засобів балансування навантаження для підвищення економічної ефективності інфраструктури, зменшення ризику деградації сервісу під час пікових навантажень і забезпечення більш прогнозованої якості обслуговування користувачів.

У роботі використано такі методи дослідження: методи теорії масового обслуговування та втратні моделі Ерланга — для формалізації процесів надходження й обслуговування запитів, оцінювання ймовірності відмови в обслуговуванні та

визначення мінімально достатньої ємності хмарного кластера; методи математичного моделювання — для побудови моделі адаптивного керування ресурсами, методу асинхронного контуру прикладного захисту та методу оцінювання ресурсної ефективності консолідації; методи теорії ймовірностей і статистичного аналізу — для опису стохастичної моделі трафіку та його характеристик, згладжування телеметрії, оцінювання середнього часу оброблення запитів і параметризації моделей; чисельні рекурентні методи — для стійкого обчислення за формулою Ерланга B та ітеративного визначення необхідної конфігурації кластера; методи системного й архітектурного аналізу — для обґрунтування архітектури без збереження стану, розділення шару даних, централізованої керованої точки входу та асинхронного контуру прикладного захисту; методи програмної інженерії — для побудови дослідницького програмного прототипу основних компонентів запропонованої інформаційної технології; методи експериментального дослідження — для підготовки та проведення перевірки працездатності розроблених моделей і архітектурних рішень у характерних сценаріях масового багатодоменного хостингу.

Ключові слова: інформаційна технологія, хмарні сервіси, хмарні обчислення, оптимізація ресурсів, консолідація ресурсів, адаптивне управління ресурсами, балансування навантаження, маршрутизація запитів, угода про рівень надання послуг, стохастична модель трафіку, асинхронний аналіз трафіку, прикладний захист, імітаційне моделювання.

ABSTRACT

Chizhov O. V. — Information technology for computing resource optimization in cloud clusters for dynamic hosting of web services — Qualification Scientific Work in the Form of a Manuscript.

Dissertation for the Degree of Doctor of Philosophy in the Specialty 122 “Computer Science” (12 — Information technologies). — National University “Kyiv Aviation Institute”, Kyiv, 2026.

The dissertation is devoted to the development and substantiation of an information technology for optimizing the use of computing resources in cloud clusters for dynamic hosting of web services. The task of large-scale servicing of a large number of low-traffic domains is considered under stochastic traffic conditions, uneven distribution of demand between domains, the dominance of encrypted connections, and strict requirements for availability and performance defined by service level agreements (SLA). It is shown that this segment is characterized by a contradiction between the aim of maximizing the consolidation of computing resources and the need to ensure predictable quality of service, in particular under application-layer availability threats.

At the same time, a specific feature of the studied class of systems is that traditional approaches to reserving computing capacity do not provide a proper balance between economic efficiency and quality of service. Static allocation of resources to separate nodes or groups of sites leads to fragmentation of the reserve and idle use of a significant part of computing capacity, while excessive density of domain placement increases the risk of mutual influence between tenants and SLA violations. In this regard, the dissertation considers not only the task of scaling as a response to overload, but also the complex task of formally determining the minimum sufficient cluster capacity, stabilizing the control loop, and maintaining system operability under a changing set of domains and certificates, as well as under changing intensity of network requests.

The object of the research is the processes of planning, dynamic allocation, and adaptive management of computing resources in cloud systems for large-scale servicing of web services.

The subject of the research is mathematical models, methods, and algorithms for optimizing the use of computing resources in a cloud cluster for dynamic web hosting, taking into account probabilistic quality-of-service requirements and the asynchronous organization of the application protection loop.

The aim of the work is to develop an information technology for optimizing the use of computing resources in cloud clusters for dynamic hosting of web services, which ensures compliance with SLA requirements under stochastic multi-domain workload conditions and creates opportunities for applying application-layer protection mechanisms without degrading the performance of the main processing path for legitimate requests.

The dissertation develops a mathematical model of adaptive capacity management for a cloud cluster of large-scale web hosting, in which the infrastructure is represented as a single pool of computing resources, while random-access memory is defined as a critically limiting resource that determines the permissible concurrency of request processing. To formalize service level requirements, the Erlang loss model is used, which made it possible to relate the intensity of stochastic multi-domain traffic, service parameters, and the minimum sufficient cluster capacity. The difference of the proposed model is that capacity management is considered not as a heuristic response to exceeding hardware metrics, but as a calculation procedure related to the probabilistic estimation of service denial. This makes it possible to move from local analysis of the load of a separate server to the assessment of the state of the entire cluster as a shared resource. Within the model, the relationship is taken into account between the number of simultaneously processed requests, the average amount of RAM per request, the available memory of computing nodes, and the acceptable blocking probability. This approach provides a formal basis for making decisions on adding nodes to the cluster or removing them from it without an unjustified increase in the reserve.

The method for determining the required configuration of a cloud cluster and managing its scaling has been improved on the basis of numerically stable recurrent computation of blocking probability, iterative selection of the minimum sufficient capacity, and mechanisms of threshold and time hysteresis, which reduce the risk of self-oscillations in the control loop

under short-term stochastic load fluctuations. The proposed method makes it possible to distinguish short-term random traffic spikes from a stable change in load that requires a change in the cluster configuration. The use of threshold hysteresis prevents excessively frequent state switching in the case of minor fluctuations in the calculated capacity, while time hysteresis ensures confirmation of the trend before scaling is performed. As a result, the control loop remains stable in scenarios typical of large-scale web hosting, where the load of individual domains may change unevenly, impulsively, and asynchronously. The practical result is a reduction in the number of unjustified scaling operations and the maintenance of service stability without excessive resource reservation.

A method of the asynchronous application protection loop has been developed. It formalizes the operation of a subsystem for delayed analysis of HTTP/HTTPS requests outside the critical service path, describes the dynamics of a multichannel system with a limited queue, the delayed verification mechanism, and the conditions for timely response to application-layer attacks. This made it possible to substantiate an approach in which resource-intensive traffic analysis does not degrade the performance of legitimate traffic processing.

The method for assessing the resource efficiency of consolidation in large-scale cloud web hosting has been further developed. It is based on an integral comparison of the traditional shared hosting architecture and a dynamic cloud cluster in terms of the amount of engaged resource over time and the consolidation efficiency coefficient.

Based on the obtained results, the architecture of a dynamic cloud cluster has been substantiated, in which computing nodes operate without maintaining state, while data storage is moved to a separate layer. Within this architecture, a centralized controlled entry point is provided for TLS connection termination, collection of operational metrics, controlled request routing, and implementation of control actions during scaling. A research software prototype has been formed, within which the main components of the proposed information technology have been implemented. These components are necessary for full-scale experimental verification of the developed models and architectural solutions. The experimental part of the work is aimed at verifying the operability of the proposed solutions in scenarios close to the

operating conditions of a multi-domain hosting environment. The scalability of the central entry point during HTTPS request processing, the operation of the capacity calculation and scaling module, the resource efficiency of the adaptive cluster approach in comparison with the traditional shared hosting model, and the system's resistance to HTTP flood load in the presence of an asynchronous application protection loop were studied. The obtained results confirm the possibility of applying the proposed information technology to increase the density of resource consolidation, reduce unproductive reservation, and maintain SLA requirements in a cloud cluster for dynamic web hosting.

The practical significance of the obtained results lies in the possibility of using them in the design and modernization of cloud web hosting infrastructure, in particular for building automatic scaling systems, determining the minimum sufficient cluster configuration, assessing the efficiency of resource consolidation, and implementing application protection without including resource-intensive analysis in the critical path of request processing. The proposed models and methods can be used by hosting providers, developers of multi-domain web platforms, and load balancing tools to improve the economic efficiency of infrastructure, reduce the risk of service degradation during peak loads, and ensure a more predictable quality of service for users.

The following research methods were used in the dissertation: methods of queueing theory and Erlang loss models, for formalizing the processes of request arrival and service, estimating the probability of service denial, and determining the minimum sufficient capacity of a cloud cluster; methods of mathematical modeling, for building the adaptive resource management model, the method of the asynchronous application protection loop, and the method for assessing resource efficiency of consolidation; methods of probability theory and statistical analysis, for describing stochastic traffic characteristics, smoothing telemetry, estimating the average request processing time, and parameterizing the models; numerical recurrent methods, for stable computation using the Erlang B formula and iterative determination of the required cluster configuration; methods of system and architectural analysis, for substantiating the stateless architecture, separation of the data layer, centralized

controlled entry point, and asynchronous application protection loop; methods of software engineering, for building a research software prototype of the main components of the proposed information technology; and methods of experimental research, for preparing and conducting verification of the operability of the developed models and architectural solutions in representative scenarios of large-scale multi-domain hosting.

Keywords: information technology, cloud services, cloud computing, resource optimization, resource consolidation, adaptive resource management, load balancing, request routing, service level agreement, SLA, stochastic traffic model, asynchronous traffic analysis, application-layer protection, simulation modeling.

СПИСОК ПУБЛІКАЦІЙ ЗДОБУВАЧА ЗА ТЕМОЮ ДИСЕРТАЦІЇ

Список публікацій, у яких опубліковано основні наукові результати дисертації

Статті у фахових наукових виданнях:

1. Chizhov A., Fesenko A. Intelligent load balancing management in cloud web hosting: Evaluation criteria and methodology. Problems of Informatization and Management. 2025. No. 3(83). P. 138–147. DOI: 10.18372/2073-4751.83.20551.
   Особистий внесок автора: розроблено систему критеріїв оцінювання балансувальників навантаження для хмарного вебхостингу, сформульовано методологію їх експериментального порівняння, визначено показники оцінювання та проаналізовано отримані результати.
2. Чижов О., Фесенко А. Метод планування потужностей для консолідації ресурсів у масовому хмарному веб-хостингу. *Проблеми інформатизації та управління*. 2025. Т. 83, № 4. С. 151–161. DOI: 10.18372/2073-4751.84.20905.
   Особистий внесок автора: розроблено математичну модель планування ємності хмарного кластера для масового вебхостингу, сформульовано умови дотримання SLA, запропоновано метод визначення мінімально достатньої конфігурації та виконано відповідні розрахунки.

Статті у періодичних виданнях, які включено до міжнародних наукометричних баз:

3. Chizhov A., Fesenko A. Web hosting companies’ client solutions: A study of a strategic standpoint [Special issue]. Corporate & Business Strategy Review. 2025. Vol. 6, No. 1. P. 421–429. DOI: 10.22495/cbsrv6i1siart18 [Scopus Q3].
   Особистий внесок автора: проведено порівняльний аналіз підходів до надання вебхостингових послуг, обґрунтовано доцільність переходу до хмарної кластерної моделі для слабо навантажених вебресурсів, виконано інтерпретацію результатів дослідження.

4. Chizhov A., Fesenko A., Ziuziun V., Basshykyzy D. Cloud shared hosting DDoS resistance and potential ways of protection. CEUR Workshop Proceedings. 2024. Vol. 3925. P. 13–23. [Scopus].

   Особистий внесок автора: обґрунтовано підхід до підвищення стійкості хмарного shared hosting до DDoS-атак типу HTTP Flood, адаптовано модель оцінювання доступності за атакувальних впливів, сформульовано архітектурні принципи асинхронного захисного контуру та виконано аналіз ефективності запропонованих рішень.

Наукові праці, які додатково відображають наукові результати дисертації:

5. Чижов О., Фесенко А., Пустосвіт М., Німченко Т. Методи оптимізації розподілу навантаження на обчислювальний ресурс інфраструктури хмарного сервісу. *Захист інформації*. 2023. Т. 25, № 4. С. 207–213. DOI: 10.18372/2410-7840.25.18226.

   Особистий внесок автора: проведення порівняльного аналізу статичних та динамічних алгоритмів балансування навантаження для оптимізації використання обчислювальних ресурсів хмарної інфраструктури.

Патент на корисну модель:

6. Спосіб захисту вебдодатків на основі асинхронного аналізу запитів: пат. 160974 U Україна : H04W 12/08 , G06F 21/00 , G06F 12/14 / Дуднік А. С., Фесенко А. О., Чижов О. В.; заявник і патентовласник ПрАТ “ВНЗ”МАУП”. № u 2025 02265; заявл. 13.05.2025; опубл. 22.10.2025, Бюл. № 43. 4 с.

   Особистий внесок автора: запропоновано технічну ідею та принцип функціонування способу захисту обчислювальних ресурсів хмарного вебхостингового кластера від DDoS-атак типу HTTP Flood, розроблено логіку асинхронного аналізу HTTP/HTTPS-запитів і механізм формування динамічного списку блокування.

Публікації у збірниках матеріалів конференцій:

7. Чижов О., Фесенко А. Застосування моделі Маркова при діагностиці інфраструктури хмарного сервісу. Проблеми кібербезпеки інформаційно-телекомунікаційних систем : збірник матеріалів доповідей та тез (м. Київ, 27 квіт. 2023 р.). Київ : Київський національний університет імені Тараса Шевченка, 2023. С. 157–158.
8. Чижов О., Фесенко А. Застосування змішаної моделі Гауса при діагностиці інфраструктури хмарного сервісу. Прикладні системи та технології в інформаційному суспільстві : збірник тез доповідей і наукових повідомлень учасників VII Міжнародної науково-практичної конференції (Київ, 29 верес. 2023 р.). За заг. ред. В. Плескач, В. Зосімов, О. Булгакова. Київ : Київський національний університет імені Тараса Шевченка, 2023. С. 276–279.
9. Чижов О., Фесенко А. SaaS cloud shared hosting як сучасне рішення для малого бізнесу, щоб бути онлайн. Інформаційна безпека та комп’ютерні технології : тези доповідей VII Міжнародної науково-практичної конференції до 30-річчя кафедри кібербезпеки та програмного забезпечення (м. Кропивницький, 1 листоп. 2023 р.). М-во освіти і науки України, Центральноукраїнський національний технічний університет, каф. кібербезпеки та програмного забезпечення. Кропивницький : ЦНТУ, 2023. С. 109–110.
10. Chizhov A. Cloud Migration Failures: Root Causes and Practical Remediation Framework. A. Chizhov, A. Fesenko. AVIA-2025: матеріали XXV Міжнародної науково-технічної конференції (Київ, 23–24 травня 2025 р.). Київ: КАІ, 2025. – Секція 16. С. 16.6–16.9.
11. Чижов О. В., Фесенко А. О. Методологія експериментальної перевірки балансувальників HTTPS у кластерних вебхостингових середовищах. Міжнародна науково-технічна конференція «Інтелектуальні технології лінгвістичного аналізу» : тези доповідей (Київ, 22–23 жовтня 2025 р.). – Київ: КАІ, 2025. – С. 96–97. – DOI: 10.18372/kai.2025.b04/itla.

ЗМІСТ

АНОТАЦІЯ 2

ABSTRACT 7

СПИСОК ПУБЛІКАЦІЙ ЗДОБУВАЧА ЗА ТЕМОЮ ДИСЕРТАЦІЇ 12

ЗМІСТ 15

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ 18

ВСТУП 19

Розділ 1. СТАН ПРОБЛЕМИ ТА ПЕРЕДУМОВИ СТВОРЕННЯ ІНФОРМАЦІЙНОЇ ТЕХНОЛОГІЇ ДИНАМІЧНОГО ХОСТИНГУ У ХМАРНОМУ КЛАСТЕРІ 28

1.1. Контекст і актуальність задачі для провайдера хостингу 29

1.2. Аналіз існуючих архітектур хостингу і їх меж застосування 37

1.3. Підходи до оптимізації використання обчислювальних ресурсів у хмарі 46

1.4. Балансування навантаження і керування TLS у багатодоменному середовищі 57

1.5. Загрози доступності та захист на рівні L7 у системах масового хостингу 65

1.6. Висновки з огляду та постановка задачі дослідження 73

1.7. Висновки до першого розділу 78

Розділ 2. МАТЕМАТИЧНІ МОДЕЛІ ПЛАНУВАННЯ ЄМНОСТІ ТА АДАПТИВНОГО УПРАВЛІННЯ РЕСУРСАМИ ХМАРНОГО КЛАСТЕРА 80

2.1. Формалізація завдання адаптивного управління ресурсами у Stateless-кластері 82

2.2. Математична модель розрахунку необхідної ємності кластера 87

2.3. Модель черг для асинхронної системи захисту (L7) 92

2.4. Метод оцінювання ресурсної ефективності адаптивного кластера 99

2.5. Висновки до другого розділу 104

Розділ 3. РОЗРОБКА ІНФОРМАЦІЙНОЇ ТЕХНОЛОГІЇ АДАПТИВНОГО ХОСТИНГУ 107

3.1. Архітектура Stateless-кластера з розділеним шаром даних 108

3.2. Алгоритмічне забезпечення підсистеми маршрутизації та оброблення запитів 115

3.3. Програмна реалізація модуля масштабування 121

3.4. Програмна реалізація методу асинхронного захисту (L7) 130

3.5. Висновки до третього розділу 139

Розділ 4. ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ІНФОРМАЦІЙНОЇ ТЕХНОЛОГІЇ 141

4.1. Програмна реалізація інформаційної технології 142

4.2. Опис експериментального стенду та параметрів середовища дослідження 148

4.3. Верифікація експериментального стенду 151

4.4. Дослідження масштабованості балансувальника у багатодоменному TLS-середовищі 155

4.5. Експериментальна перевірка модуля розрахунку ємності кластера та механізму масштабування 162

4.6. Порівняльне дослідження ресурсної ефективності класичного режиму спільного хостингу та адаптивного кластерного підходу 172

4.7. Дослідження стійкості кластера до HTTP flood-атак та внеску асинхронного контуру захисту 181

4.8. Висновок до четвертого розділу 187

ВИСНОВКИ 189

Список використаних джерел 191

Додаток 1. Ключові елементи модуля prediction балансувальника (ILB) 204

Додаток 2. Ключові елементи модуля scaling балансувальника (ILB) 210

Додаток 3. Ключові елементи модуля cluster балансувальника (ILB)..... 214

Додаток 4. Ключові елементи модуля acl балансувальника (ILB) 219

Додаток 5. Скрипт побудови сертифікатів 223

Додаток 6. Скрипт k6 емулятора навантаження для тестування backend серверів 231

Додаток 7. Тижневий нормалізований профіль навантаження 241

Додаток 8. Документи впровадження основних результатів дисертаційної роботи 242

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

ACL — списки доступу безпеки

API — програмний інтерфейс застосунку

CMS — система керування контентом

CPU — центральний процесор

DNS — система доменних імен

ECDSA — алгоритм цифрового підпису на еліптичних кривих

GPU — графічний процесор

HTTP — протокол передавання гіпертексту

HTTPS — захищений протокол передавання гіпертексту

ILB — інтелектуальний балансувальник навантаження

IP — інтернет-протокол

ML — машинне навчання

RAM — оперативна пам’ять

RPS — кількість запитів за секунду

RSS — обсяг фізичної оперативної пам’яті, що використовується процесом

SLA — показник якості обслуговування

SSL — протокол захищених сокетів

TCP — протокол керування передаванням

TLS — Безпека транспортного рівня

TTFB — час до отримання першого байта відповіді

VPC — віртуальна приватна хмара

VPS — віртуальний приватний сервер

ВСТУП

Актуальність теми. Сучасний ринок масового вебхостингу як сегмент інформаційно-комунікаційних систем постає перед складною інфраструктурною проблемою — як одночасно обслуговувати десятки тисяч слабо навантажених сайтів, гарантувати кожному з них жорсткі вимоги щодо рівня надання послуг (SLA) і водночас тримати ціну оренди на низькому рівні. Ситуація ускладнюється стохастичною природою трафіку, попит на інформаційні послуги залежить від часу доби та дня тижня. Такі чинники як нерівномірний попит між доменами та глобальний перехід на ресурсоємні захищені з'єднання (TLS/SSL), також ускладнюють виконання цього архітектурного завдання.

Класична модель спільного хостингу історично вирішувала проблему щільності розміщення завдяки розподілу ресурсів одного фізичного сервера між багатьма вебсайтами. Однак під час короткочасних пікових навантажень чи аномальної активності окремих ресурсів, ця архітектура виявляє критичну системну вразливість, що проявляється у негативному впливі орендарів один на одного. Для згладжування цього ефекту доводиться утримувати значний статичний резерв потужності на кожному вузлі. Як наслідок, більшу частину часу ці ресурси просто простоюють, що зводить нанівець спроби підвищити фактичну щільність консолідації та суттєво погіршує економічну ефективність інфраструктури. Зменшуючи резерв та збільшуючи щільність розміщення, провайдер послуг покращує економічну складову, але виникає ризик порушення вимог SLA. Жорстке дотримання вимог якості, призводить до збільшення резерву, який більшу частину часу не використовується.

У реаліях масового багатодоменного вебхостингу, у багатьох практичних сценаріях, обмежувальним ресурсом є не стільки процесорний час, скільки оперативна пам'ять (RAM). Саме оперативна пам'ять визначає граничну конкурентність оброблення запитів, і саме дефіцит пам'яті викликає накопичення черг, перехід до підкачки, деградацію часових характеристик і ризик каскадних відмов. Зважаючи на це, завдання оптимізації використання обчислювальних ресурсів не може зводитися лише до

спостереження за усередненими апаратними метриками. Вона потребує формалізованого зв’язку між інтенсивністю вхідного трафіку, рівнем конкурентності, доступною ємністю системи та вимогами SLA.

Перспективним напрямком розв’язання цієї проблеми є перехід до кластерної організації інфраструктури, у якій серверні вузли розглядаються як елементи спільного пулу ресурсу, а сумарна ємність може змінюватися відповідно до поточного стану системи. Проте впровадження спільного пулу саме по собі не розв’язує проблеми. Ефективність такого підходу залежить від наявності математичних моделей і алгоритмів керування, здатних працювати в умовах стохастичного навантаження, короткочасних пікових сплесків і безперервної зміни конфігурації системи. Динамічна конфігурація є штатною властивістю експлуатації масового вебхостингу та пов’язана з постійним додаванням і видаленням доменних імен та періодично масовою ротацією TLS/SSL-сертифікатів. Це висуває додаткові вимоги до моделей і засобів керування ресурсами, оскільки вони мають забезпечувати адаптивність системи без порушення її доступності та показників якості обслуговування.

Додаткову складність створюють загрози доступності на прикладному рівні. Атаки класу HTTP flood та інші автоматизовані сценарії шкідливої активності якісно маскуються під легітимні HTTP/HTTPS-запити. За умов масового використання TLS такі загрози неможливо надійно виявляти на нижчих рівнях мережевої моделі без термінування захищеного з’єднання та аналізу вмісту HTTP-запиту. Включення ресурсоємних перевірок безпосередньо до синхронного тракту оброблення запитів призводить до зростання затримок для звичайних користувачів та підвищує ризик невиконання показників рівня обслуговування (SLA).

З огляду на викладене, виникає об’єктивна потреба у розробленні комплексної інформаційної технології. Вона має поєднати в собі математичне планування ємності, адаптивне керування ресурсами хмарного кластера і, як наслідок, підвищення ефективності використання обчислювальних ресурсів. Архітектурні засади організації асинхронного контуру прикладного захисту, що дозволять протидіяти атакам без

деградації швидкодії оброблення легітимних запитів. Саме розв'язанню цієї актуальної науково-прикладної задачі присвячено дисертаційну роботу.

Зв'язок роботи з науковими програмами, планами, темами. Дисертаційну роботу виконано в Національному університеті «Київський авіаційний інститут» відповідно до планів науково-дослідної роботи кафедри комп'ютерних інформаційних технологій. Тематично дисертаційне дослідження пов'язане з напрямами наукових робіт, присвячених хмарним обчисленням, оптимізації використання обчислювальних ресурсів, балансуванню навантаження, забезпеченню доступності веб-сервісів та інформаційній безпеці хмарних інфраструктур в умовах прикладних атак. Особисто автором у межах зазначеного напряму виконано дослідження, пов'язані з розробленням математичних моделей планування ємності хмарного кластера, методу адаптивного масштабування, моделі асинхронного контуру прикладного захисту та архітектурних засад динамічного хостингу вебсервісів.

МЕТА І ЗАВДАННЯ ДОСЛІДЖЕННЯ.

Мета дисертаційної роботи: розробити інформаційну технологію оптимізації використання обчислювальних ресурсів у хмарних кластерах для динамічного хостингу веб-сервісів, яка забезпечує дотримання вимог угод про рівень надання послуг в умовах стохастичного багатодоменного навантаження та створює умови для застосування прикладних механізмів захисту без погіршення швидкодії основного тракту оброблення легітимних запитів.

Для досягнення поставленої мети в роботі необхідно розв'язати такі завдання:

1. Провести аналіз сучасних архітектурних підходів до організації масового вебхостингу, методів консолідації ресурсів, масштабування та засобів прикладного захисту, а також виявити обмеження їх застосування в умовах багатодоменного середовища.
2. Розробити математичну модель планування ємності хмарного кластера для динамічного вебхостингу, яка встановлює зв'язок між стохастичною моделлю вхідного трафіку, рівнем конкурентності оброблення запитів, обмеженнями

оперативної пам’яті та ймовірністю відмови в обслуговуванні, пов’язаною з вимогами щодо рівня надання послуг (SLA).

3. Розробити математичну модель асинхронного контуру прикладного захисту, яка формалізує функціонування підсистеми відкладеного аналізу HTTP/HTTPS-запитів поза критичним шляхом обслуговування, описує динаміку черги, механізм відкладеної верифікації та визначає умови своєчасної реакції на прикладні атаки.
4. Удосконалити метод визначення необхідної ємності кластера і керування його масштабуванням на основі чисельно стійкого розрахунку ймовірності блокування та механізмів стабілізації керувального контуру за стохастичних коливань навантаження, а також розробити методичний підхід до кількісного оцінювання ресурсної ефективності консолідації в хмарному кластері порівняно з традиційною моделлю статичного розміщення на спільному хостингу.
5. Обґрунтувати архітектурні принципи побудови динамічного хмарного кластера без збереження стану на обчислювальних вузлах, з розділеним шаром даних та централізованою керованою точкою входу, реалізувати програмний прототип основних компонентів запропонованої інформаційної технології та виконати експериментальну перевірку працездатності розроблених моделей, методів і архітектурних рішень у характерних сценаріях багатодоменного динамічного вебхостингу.

Об’єктом дослідження є процеси планування, динамічного розподілу та адаптивного керування обчислювальними ресурсами в хмарних системах масового обслуговування вебсервісів.

Предметом дослідження є математичні моделі, методи та алгоритми оптимізації використання обчислювальних ресурсів у хмарному кластері динамічного вебхостингу з урахуванням імовірнісних вимог до якості обслуговування та асинхронної організації контуру прикладного захисту.

МЕТОДИ ДОСЛІДЖЕННЯ

Для виконання поставлених завдань у дисертаційній роботі застосовано комплексний методологічний апарат:

- **методи теорії масового обслуговування та втратні моделі Ерланга** — для формалізації процесів надходження й обслуговування запитів, оцінювання ймовірності відмови та визначення мінімально достатньої ємності хмарного кластера;
- **методи математичного моделювання, теорії ймовірностей та статистичного аналізу** — для опису стохастичних характеристик багатодоменного трафіку й оцінювання параметрів функціонування системи;
- **чисельні рекурентні методи** — для забезпечення стійкого обчислення показників якості обслуговування (SLA);
- **методи системного й архітектурного аналізу** — під час обґрунтування принципів побудови динамічного хмарного кластера, організації централізованої точки входу та проєктування асинхронного контуру прикладного захисту;
- **методи програмної інженерії, імітаційного моделювання та експериментального дослідження** — для розроблення дослідницького програмного прототипу й верифікації працездатності створених моделей і алгоритмів у реалістичних сценаріях масового динамічного вебхостингу.

НАУКОВА НОВИЗНА ОДЕРЖАНИХ РЕЗУЛЬТАТІВ.

Уперше розроблено математичну модель адаптивного керування ємністю хмарного кластера масового вебхостингу, яка, на відміну від підходів статичного резервування, розглядає інфраструктуру як єдиний обчислювальний пул і визначає оперативну пам’ять як критично обмежувальний ресурс. Це дало змогу формалізувати вимоги SLA через цільову ймовірність відмови в обслуговуванні за формулою Ерланга B та визначати мінімально достатню ємність кластера в умовах стохастичного багатодоменного трафіку.

Уперше розроблено математичну модель асинхронного контуру прикладного захисту, яка, на відміну від синхронних схем інспекції трафіку, описує підсистему відкладеного аналізу HTTP/HTTPS-запитів поза критичним шляхом обслуговування як багатоканальну систему з обмеженою чергою. Модель враховує механізм відкладеної верифікації та умови своєчасного реагування на атаки, що дає змогу поєднати вимоги безпеки зі збереженням швидкодії легітимного трафіку.

Удосконалено метод визначення необхідної конфігурації хмарного кластера та керування його масштабуванням шляхом використання чисельно стійкої рекурентної процедури розрахунку ймовірності блокування, ітеративного підбору мінімально достатньої ємності та механізмів порогового і часового гістерезису. Це зменшує ризик автоколивань системи під час короткочасних флуктуацій навантаження.

Набув подальшого розвитку метод оцінювання ресурсної ефективності консолідації в масовому вебхостингу, яка ґрунтується на інтегральному порівнянні фрагментованої архітектури та динамічного кластера за функціями загальних витрат ресурсу, а не на локальних оцінках завантаження окремих вузлів. Це дає змогу кількісно обґрунтовувати доцільність переходу до кластерної моделі за умови дотримання SLA.

Сукупність розроблених моделей та методів становить наукову основу інформаційної технології оптимізації використання обчислювальних ресурсів у хмарних кластерах для динамічного хостингу веб-сервісів.

ПРАКТИЧНЕ ЗНАЧЕННЯ ОДЕРЖАНИХ РЕЗУЛЬТАТІВ.

Практичне значення одержаних результатів полягає у створенні науково-методичної основи для побудови та модернізації інфраструктур масового вебхостингу. Розроблені математичні моделі, метод і архітектурні рішення дають можливість підвищити щільність консолідації обчислювальних ресурсів, гарантуючи водночас дотримання угод про рівень надання послуг (SLA) та стійкість до прикладних кібератак.

Застосування одержаних результатів на практиці дає змогу точно визначати мінімально достатню ємність хмарного кластера для багатодоменного навантаження, обґрунтовувати правила адаптивного масштабування без ризику автоколивань системи та кількісно оцінювати ефект від переходу до динамічної кластерної моделі. Це суттєво зменшує обсяги непродуктивного резервування потужностей, підвищує економічну ефективність інфраструктури та мінімізує операційні ризики під час пікових навантажень. Практичну цінність має також запропонований підхід до організації асинхронного контуру прикладного захисту. Такий підхід дозволяє інтегрувати ресурсоємний аналіз HTTP/HTTPS-запитів поза критичним шляхом обслуговування та поєднати вимоги безпеки з вимогами швидкості. На основі одержаних результатів обґрунтовано архітектуру динамічного хмарного кластера без збереження стану на обчислювальних вузлах із розділеним шаром даних та централізованою керованою точкою входу, яка функціонально забезпечує термінування TLS-з’єднань, моніторинг і збирання телеметрії, маршрутизацію запитів, застосування списків блокування та реалізацію керувальних сигналів масштабування.

Результати дисертаційної роботи можуть бути використані хостинговими компаніями, постачальниками хмарних сервісів та хмарної інфраструктури, розробниками багатодоменних вебплатформ, а також веб-сервісів із підвищеними вимогами до доступності, зокрема пов’язаних із критичною інфраструктурою. Практичне застосування одержаних результатів охоплює проєктування систем автоматичного масштабування, вибір цільової архітектури вебхостингового середовища, оцінювання економічної доцільності консолідації ресурсів і побудову засобів

асинхронного аналізу трафіку для протидії прикладним атакам без погіршення часових характеристик обслуговування. Одержані результати також можуть бути використані в навчальному процесі під час викладання дисциплін, пов’язаних із хмарними обчисленнями, архітектурою розподілених систем, балансуванням навантаження, масштабуванням хмарних сервісів, інформаційною безпекою та кіберзахистом веб-сервісів.

Результати дисертаційної роботи використано в ТОВ «АксонСофт» для аудиту поточної ефективності використання серверного парку компанії, а також у навчальному процесі Національного університету «Київський авіаційний інститут» під час викладання дисциплін «Хмарні технології» та «Адміністрування комп’ютерних систем та мереж». Факт використання підтверджено відповідними документами у додатку 8.

ОСОБИСТИЙ ВНЕСОК ЗДОБУВАЧА.

Наукові результати, викладені в дисертаційній роботі та винесені на захист, одержано здобувачем самостійно. У тексті дисертації використано виключно ті результати опублікованих у співавторстві праць, що належать особисто авторові (детальний опис особистого внеску наведено у списку публікацій).

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДИСЕРТАЦІЇ.

Основні положення, результати та висновки дисертаційної роботи доповідались і обговорювались на міжнародних науково-технічних та науково-практичних конференціях: «Проблеми кібербезпеки інформаційно-телекомунікаційних систем» (м. Київ, 2023); VII Міжнародній науково-практичній конференції «Прикладні системи та технології в інформаційному суспільстві» (м. Київ, 2023); VII Міжнародній науково-практичній конференції «Інформаційна безпека та комп’ютерні технології» (м. Кропивницький, 2023); XXV Міжнародній науково-технічній конференції AVIA-2025 (м. Київ, 2025); Міжнародній науково-технічній конференції «Інтелектуальні технології лінгвістичного аналізу» (м. Київ, 2025). Окрім того, результати дисертаційної роботи були обговорені на науково-технічних семінарах кафедри комп’ютерних інформаційних

технологій, а також на науково-практичних конференціях молодих учених і студентів «ПОЛІТ. Сучасні проблеми науки», які проводилися в Державному університеті «Київський авіаційний інститут».

ПУБЛІКАЦІЇ.

Основні наукові положення та результати дисертації висвітлено у 10 публікаціях, серед яких: 2 статті у фахових виданнях України, 2 статті у виданнях, що індексуються наукометричною базою Scopus, 1 праця, що додатково відображає результати дослідження, та 5 публікацій у матеріалах міжнародних конференцій. Вагомим практичним здобутком за темою дисертації є одержання патенту України на корисну модель (№ 160974 U).

СТРУКТУРА ТА ОБСЯГ ДИСЕРТАЦІЇ.

Дисертація складається з анотації, вступу, чотирьох розділів, загальних висновків, списку використаних джерел та додатків. Повний обсяг роботи становить 243 сторінка, з них 172 сторінки основного тексту. Ілюстративний матеріал містить 31 рисунок і 15 таблиць; додатково наведено 8 додатків. Список використаних джерел налічує 114 найменувань.

РОЗДІЛ 1. СТАН ПРОБЛЕМИ ТА ПЕРЕДУМОВИ СТВОРЕННЯ ІНФОРМАЦІЙНОЇ ТЕХНОЛОГІЇ ДИНАМІЧНОГО ХОСТИНГУ У ХМАРНОМУ КЛАСТЕРІ

У цьому розділі сформовано теоретико-прикладну основу дисертаційного дослідження та обґрунтовано актуальність завдання оптимізації використання обчислювальних ресурсів у хмарних кластерах для динамічного хостингу веб-сервісів. У фокусі розгляду перебувають масові сценарії хостингу слабо навантажених сайтів, для яких критичними є узгоджене поєднання трьох вимог: економічно ефективна консолідація ресурсу; дотримання показників якості обслуговування (SLA); стійкість до загроз доступності на прикладному рівні.

У такому середовищі ключову роль відіграє не лише вибір архітектури обчислювальної інфраструктури, а й організація контурів керування, що дають змогу адаптувати ємність кластера до змін інтенсивності трафіку та одночасно мінімізувати непродуктивні витрати ресурсу.

Логіку викладу матеріалу в цьому розділі підпорядковано послідовному переходу від аналізу наявних галузевих практик до строгої формалізації проблеми. Дослідження розпочинається з критичного огляду типових архітектур вебхостингу, де визначаються межі їхньої застосовності для багатодоменних середовищ. Наступним кроком систематизуються сучасні підходи до планування ємності та масштабування. Водночас особливий акцент робиться на ролі інтелектуального балансувальника навантаження як центрального вузла, що відповідає за термінування TLS, збирання телеметрії та ініціацію керувальних впливів. Оскільки масовий хостинг є вразливим до специфічних кіберзагроз, окремий фокус спрямовано на атаки прикладного рівня (зокрема HTTP flood). Розглянемо практичні обмеження традиційного синхронного аналізу запитів, що створюють ризики деградації сервісу. Синтез цих результатів дає змогу наприкінці розділу чітко сформулювати проблемну ситуацію та обґрунтувати постановку завдань дослідження, які надалі розв’язуватимуться математичними (розділ 2) та архітектурними (розділ 3) методами.

1.1. Контекст і актуальність задачі для провайдера хостингу

Сучасний сегмент хмарних сервісів розміщення веб-ресурсів, як складова інформаційно-комунікаційних систем, визначається не лише розвитком великих корпоративних інформаційних систем. Відчутну частку попиту формує масовий клас слабо навантажених вебсайтів [1,2], до якого належать вебплатформи малого бізнесу, регіональні інформаційні ресурси, проєкти громад та організацій, а також персональні вебсторінки [1,3]. Характерною ознакою таких ресурсів є відносно стала структура та обмежена аудиторія, активність якої здебільшого зводиться до нерегулярних сплесків, спровокованих певними подіями чи інфоприводами [4]. У термінах теорії масового обслуговування це означає, що вхідний потік запитів для цього класу систем має виражений стохастичний характер. У табл. 1.1 та на рис. 1.1 наведено статистику розподілу світового трафіку.

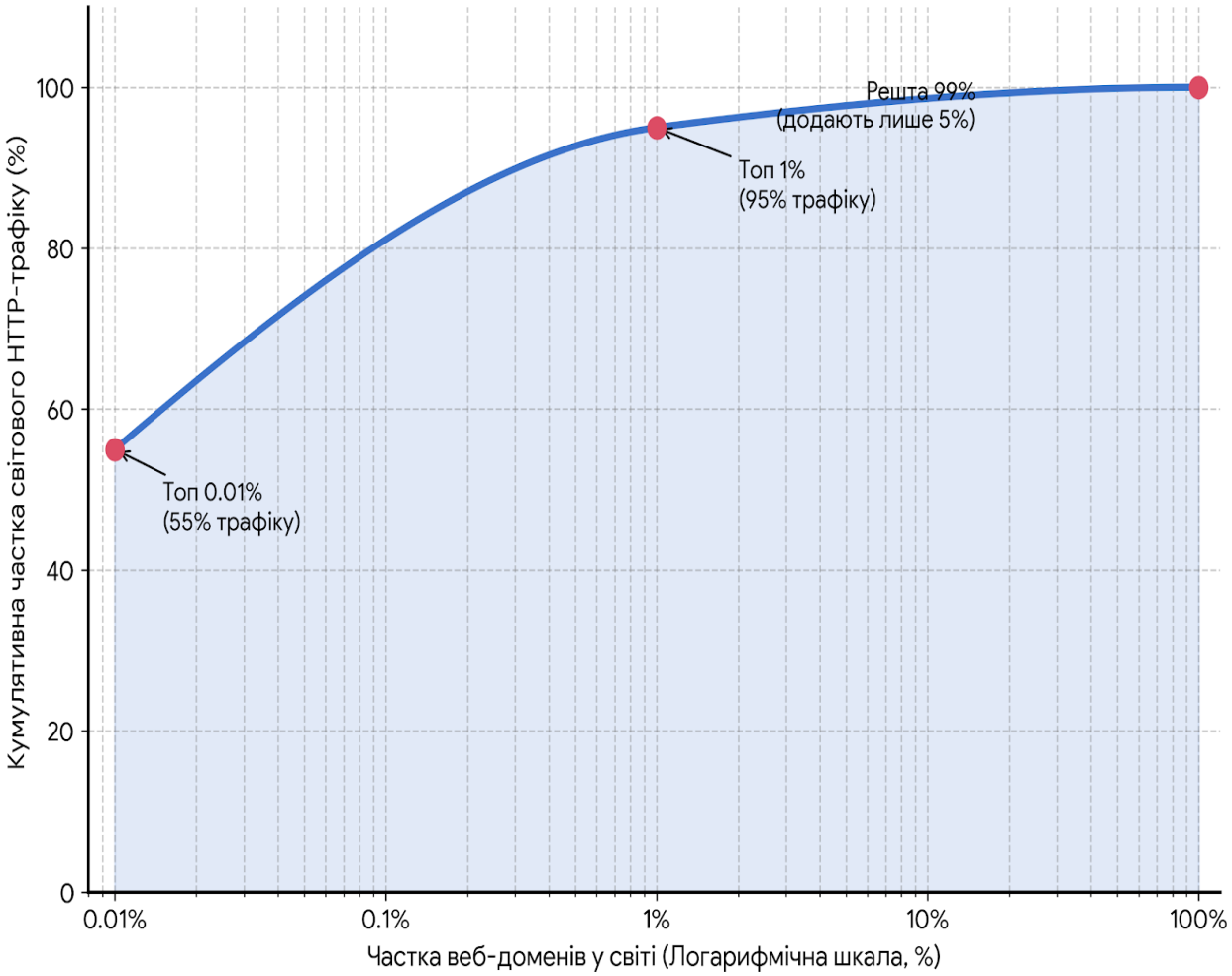

Рисунок 1.1 - Статистика розподілення світового трафіку [2,5].

Навантаження розподіляється між доменами нерівномірно. Більшість доменів упродовж тривалих інтервалів генерує мінімальну кількість запитів і практично не впливає на сумарне споживання ресурсів [1,5].

Таблиця 1.1 - Статистика розподілення світового трафіку [6,7,5].

Категорія сайтів	**Кількість сайтів (прибл.)**	**Частка від загальної кількості**	**Частка світового трафіку**	**Середнє навантаження**
High-Traffic(Top 10k)	~10 000	< 0.005%	~55%	> 10 000 RPS
Medium-Traffic(Top 1M)	~1 000 000	~0.5%	~35%	10–1000 RPS
Low-Traffic(Long Tail)	~200 000 000+	~99.5%	~10%	0,1–0,5 RPS

Проте невелика частка ресурсів здатна генерувати раптові сплески трафіку, які тимчасово стають піковими для всієї інфраструктури. Через це статистична картина попиту виявляється вкрай асиметричною. Звичайні середні показники лише маскують реальний стан речей, оскільки не беруть до уваги рідкісні, але критичні відхилення. Саме тому виникає гостра потреба в моделюванні, у якому враховано перевагу малих значень трафіку для більшості доменів та можливість поодиноких інтенсивних навантажень [8,4,9].

Для хостинг-провайдера така специфіка трансформується у складний інженерний виклик: як на спільній апаратній базі підтримувати десятки тисяч незалежних доменів [1,6,7]. Як свідчить практика експлуатації, середнє навантаження типового сайту в цьому сегменті рідко перевищує десяті частки запиту за секунду [1]. За таких умов вирішальним стає не стільки пікова пропускна здатність для окремого ресурсу, скільки здатність інфраструктури ефективно консолідувати велику кількість малих навантажень, водночас зберігаючи прийнятні показники якості обслуговування для всіх орендарів [8,10]. Історично в цьому сегменті домінує модель віртуального хостингу, що ґрунтується на спільному використанні обчислювальних ресурсів [1,11,12]. Стійке зростання цього ринку (рис. 1.2) підтверджує актуальність досліджень в області подальшого покращення масштабування та оптимізації інфраструктури [12].

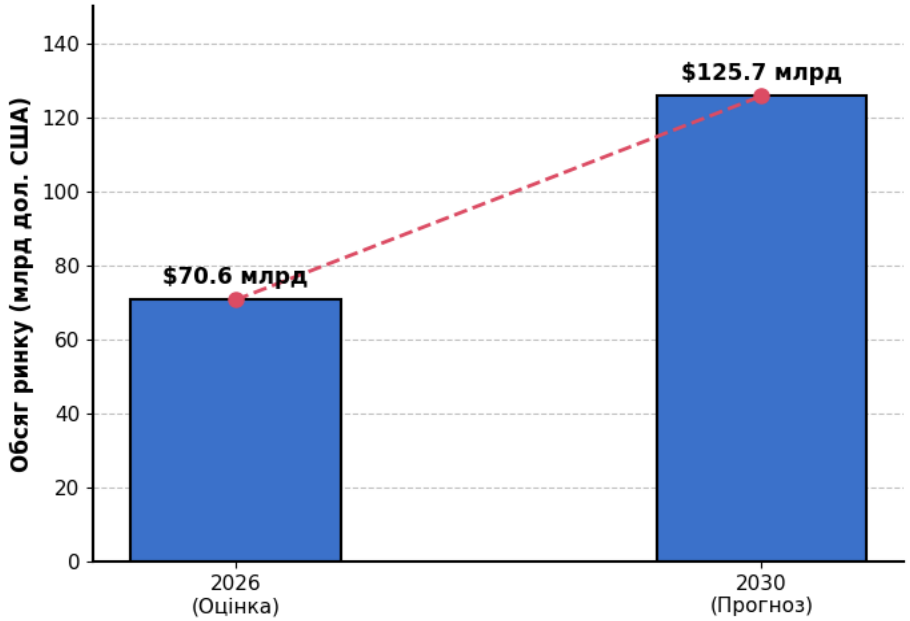

Рисунок 1.2 - Прогноз динаміки зростання глобального ринку послуг спільного хостингу на період 2026–2030 рр. [12].

Поширення моделі спільного хостингу пояснюється простотою адміністрування та високою щільністю розміщення вебресурсів на одному фізичному сервері [1]. Однак така архітектура має принципове обмеження. За умов зростання навантаження на один із сайтів виникає негативний вплив на сусідні вебсайти, оскільки вони споживають спільні ресурси процесора, пам’яті та диска. У результаті ресурсоємний сайт може витісняти інші з точки зору доступного часу процесора, оперативної пам’яті або пропускної здатності підсистеми введення-виведення [13]. Це проявляється у збільшенні часу відповіді, зростанні черг запитів та появі відмов для інших вебсайтів, що розміщені на тому самому сервері [14]. Такий локальний інцидент навантаження в межах одного орендаря здатний знижувати продуктивність і доступність групи незалежних ресурсів, що суперечить вимозі якості обслуговування [10].

Попри низьку середню інтенсивність трафіку, вимоги до якості обслуговування для цього сегмента залишаються високими. Для провайдера вони набувають формалізованого вигляду у цільових показниках доступності сервісу та гранично допустимого часу відповіді. Такі показники фіксуються в угодах про рівень надання послуг (SLA). У межах цих угод провайдер бере на себе зобов’язання підтримувати

визначений рівень працездатності інфраструктури та забезпечувати прогнозовану швидкодію під час типових сценаріїв доступу [15,16]. Проблематика полягає в тому, що навіть поодинокі пікові сплески навантаження здатні непропорційно сильно вплинути на загальне сприйняття сервісу [17]. У моменти перевантаження зростає час встановлення з'єднання та оброблення запитів, збільшується ймовірність повторних спроб і відмов, погіршується стабільність відображення сторінок [15]. Для слабо навантажених ресурсів це є критичним — користувачі очікують майже миттєвої відповіді. Відхилення від звичного часу завантаження дуже помітні навіть за незначного зростання затримок [17]. Унаслідок цього навіть короткочасна деградація продуктивності в пікові моменти активності може призвести до значних репутаційних та фінансових втрат для власника вебсайту [18].

Зазначена проблема додатково посилюється широким переходом вебсередовища на захищені протоколи передавання даних [19,20]. Використання HTTPS змінює профіль витрат на оброблення трафіку та підвищує вимоги до обчислювальних ресурсів на вході системи [21]. Встановлення захищеного з'єднання передбачає виконання криптографічних операцій, оброблення сертифікатів і перевірку параметрів сесії, зростає кількість етапів мережевого узгодження між клієнтом і сервером. Сукупно це збільшує час до отримання першого байта відповіді (TTFB) та підвищує чутливість сервісу до короткочасних перевантажень, що вкрай негативно впливає на очікуванні клієнта та якість взаємодії з вебресурсом [17,18,22].

Економічна ефективність масового вебхостингу безпосередньо залежить від здатності провайдера забезпечити високу щільність консолідації ресурсів без порушення показників доступності та швидкодії [23]. У цьому контексті важливо звернути увагу на те, який саме ресурс стає обмежувальним у практичних режимах роботи. Дослідження архітектур сучасних рішень у сфері масового хостингу засвідчують, що для типових сценаріїв дефіцитним ресурсом частіше є не процесорний час, а оперативна пам'ять (RAM). Такий висновок має прикладне значення, оскільки саме оперативна пам'ять визначає межу консолідації сайтів на одному вузлі та впливає на економіку сервісу

[24,25]. Обсяг RAM безпосередньо пов’язаний із граничною конкурентністю оброблення запитів. Кожне активне обслуговування потребує виділення пам’яті для процесів вебсервера, інтерпретатора або прикладного середовища, а також для супутніх структур даних, які використовуються під час виконання запиту [24].

У традиційній моделі розміщення вебсайтів провайдер вимушений підтримувати на кожному вузлі значний постійний запас пам’яті. Такий запас використовується як резерв для компенсації можливих сплесків активності окремих сайтів та для зменшення ризику перевантаження [24,25,23]. На практиці пікові режими виникають епізодично і для різних доменів не збігаються в часі [4,9]. Це означає, що суттєва частина зарезервованої пам’яті протягом більшої частини часу залишається невикористаною. За результатами власного моделювання процесів консолідації ресурсів, статична стратегія резервування призводить до економічно невиправданого простоювання апаратних потужностей [8]. У підсумку знижується фактична щільність розміщення сайтів та зростає вартість обслуговування одного домену.

З огляду на описані обмеження, закономірним інженерним кроком є відмова від статичного розміщення на користь кластерної архітектури з динамічним розподілом потужностей [8,23]. У такій системі фізичні сервери перестають бути ізольованими одиницями, їхні процесорні, пам’яттєві та мережеві ресурси зливаються в єдиний обчислювальний пул [26]. У режимі кластеризації резерв обчислювальної потужності не утримується у фіксованому обсязі на кожному фізичному сервері, натомість він формується за потребою та вивільняється після зменшення навантаження [27,16]. Відповідні рішення приймаються засобами балансування навантаження та автоматичного масштабування, які враховують фактичну кількість активних з’єднань, обсяг використаної оперативної пам’яті та інші експлуатаційні показники [27,23]. Такий підхід дає змогу перейти від статичного резервування до керованого використання спільних ресурсів, що особливо ефективно для середовищ із нерівномірним і епізодичним навантаженням. Оскільки пікові режими для різних доменів зазвичай не збігаються в часі, спільний пул ресурсів дає змогу використовувати резерви

ефективнішу та зменшувати частку простою [4,9]. За таких умов головне науково-практичне завдання при цьому зводиться до коректної параметризації алгоритмів керування, що дає можливість максимізувати утилізацію обладнання, неухильно дотримуючись жорстких вимог SLA щодо доступності та часу відповіді [23,16].

Поряд із завданням раціонального розподілу потужностей, критичним викликом кібербезпеки для масового хостингу стає захист інфраструктури від кібератак прикладного рівня (L7) [28]. Оскільки дрібні вебресурси здебільшого розгортаються на базі типових CMS зі стандартними конфігураціями, вони утворюють ідеальну мішень для масового автоматизованого сканування [3,29]. У цьому контексті зловмисників цікавить не стільки конкретний сайт як цінний актив, скільки сам факт його існування та можливість автоматизованої експлуатації для порушення доступності чи несанкціонованого проникнення. Найбільшу загрозу в таких середовищах становлять шторми HTTP-запитів (HTTP flood) та масові спроби підбору облікових даних (Brute Force) [30,31,32]. Їхня небезпека для багатодоменного хостингу полягає в асиметрії впливу: генерація потоку фейкових запитів, що імітують легітимну поведінку, обходиться зловмиснику вкрай дешево, тоді як система витрачає на їх оброблення дорогоцінні спільні ресурси [32]. Обидва сценарії небезпечні саме для середовищ із високою щільністю розміщення сайтів, оскільки локальна атака на один домен може призводити до непропорційного споживання спільних ресурсів і викликати деградацію якості обслуговування для сусідніх вебсайтів [30,10].

Ключова особливість зазначених загроз полягає в тому, що зловмисні запити мають синтаксично коректну структуру та формально відповідають протоколам взаємодії [28,33]. Окрім того, вони передаються через захищені канали, що унеможливлює їх аналіз на рівні мережевих пристроїв без виконання розшифрування (рис. 1.3). За таких умов традиційні засоби виявлення вторгнень, орієнтовані на нижчі рівні мережевої моделі, здебільшого фіксують лише факт наявності трафіку та базові параметри з’єднань. Вони не мають достатньої інформації для розрізнення легітимної взаємодії та шкідливої активності, яка маскується під нормальні запити прикладного

протоколу [19,34]. Це обмежує ефективність класичних підходів і обґрунтовує необхідність застосування механізмів аналізу та фільтрації саме на прикладному рівні, з урахуванням контексту запитів і поведінкових характеристик клієнтів [28,33].

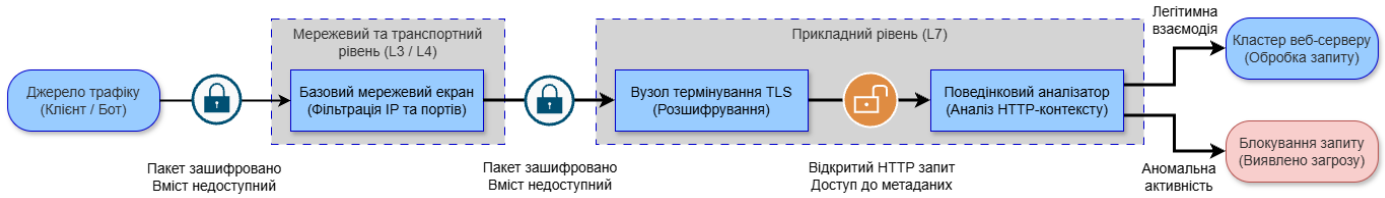

Рисунок 1.3 - Структурна схема обмежень інспектування зашифрованих пакетів на мережевому рівні та їх глибокого аналізу на прикладному рівні.

Ідеться не лише про перевірку формальної коректності протоколу, а про оцінювання ознак, що відображають зміст звернення, послідовність дій, частоту повторень, зміну параметрів сесії та інші елементи, які відрізняють автоматизовану активність від звичайної взаємодії користувача із вебресурсом [28,33].

Проте інтеграція ресурсомістких процедур аналізу безпосередньо до синхронного тракту оброблення запитів спричиняє нову архітектурну проблему. Кожна додаткова перевірка збільшує обчислювальні витрати на кожен запит та підвищує затримку відповіді для легітимного трафіку (рис. 1.4) [35,22]. За умов, коли оброблення HTTPS вже містить додаткові витрати, навіть помірне зростання часу виконання на вході системи може призводити до накопичення черг, збільшення часу очікування та погіршення показників SLA. Як наслідок, механізм захисту сам провокує деградацію сервісу для добросовісних клієнтів, що є категорично неприйнятним [15,16,21,14].

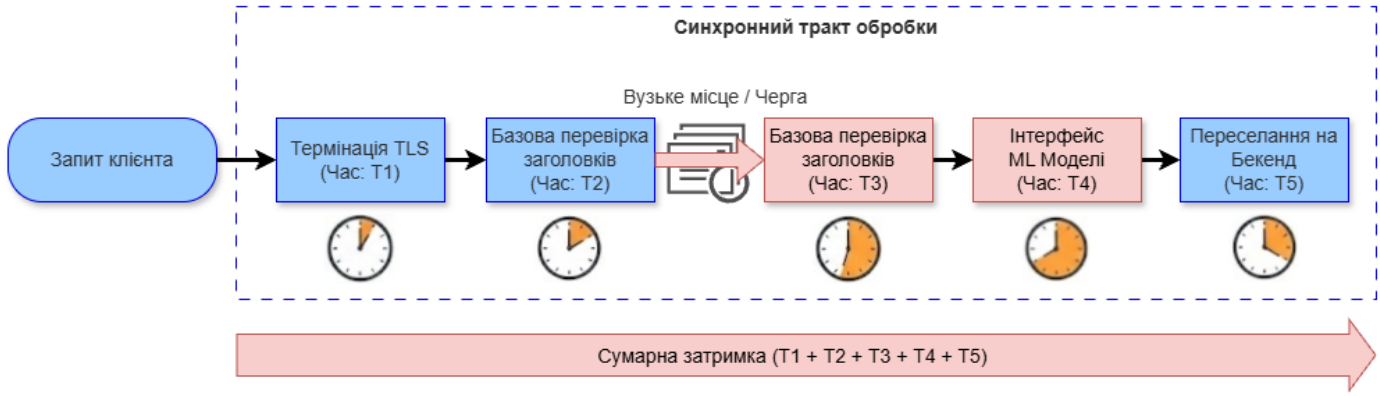

Рисунок 1.4 - Схема формування сумарної затримки в синхронному тракті обробки.

Отже, головна науково-технічна проблема для провайдера масового хостингу полягає у необхідності одночасного задоволення трьох конфліктних вимог [23]. По-перше, потрібні підвищення щільності консолідації вебресурсів і зменшення частки невикористаних апаратних потужностей [8]. По-друге, має бути забезпечене виконання показників доступності та швидкодії, визначених у SLA, з прогнозованою поведінкою системи у пікових режимах [16]. По-третє, необхідно підвищити стійкість інфраструктури до атак на прикладному рівні, не створюючи додаткових затримок для легітимного трафіку [35,28]. Додаткову складність становить те, що всі ці вимоги реалізовуються в умовах постійної зміни конфігурацій: регулярні зміни складу доменів, сертифікатів, таблиць маршрутизації. Вирішення цих інженерних викликів потребує таких методів і засобів керування, які зберігають ефективність за умови неперервної зміни стану середовища та дають можливість масштабуватися до рівня масового хостингу.

Наведені положення обґрунтовують актуальність цього дослідження та визначають підстави для подальшої постановки завдання оптимізації. З огляду на поєднання вимог до економної консолідації ресурсів, дотримання показників SLA та стійкості до прикладних атак, виникає потреба у формалізації відповідних процесів. Таке формалізоване подання має спиратися на математичні моделі, що відображають характеристики вхідного трафіку, обмеження апаратних ресурсів, умови функціонування керувальних механізмів і критерії якості обслуговування. Надалі, це дає основу для розроблення алгоритмів розподілу та масштабування ресурсів, а також підходів до організації захисних перевірок без погіршення продуктивності легітимного трафіку. Відповідні моделі, методи та алгоритми становлять зміст наступних розділів роботи, де вони розглядаються детально та в логічному зв’язку з поставленим завданням.

1.2. Аналіз існуючих архітектур хостингу і їх меж застосування

Архітектурний підхід до організації вузлів обслуговування визначає спосіб розміщення вебсервісів і правила використання спільних обчислювальних ресурсів. Від нього залежать рівень ізоляції між клієнтами та можливість інфраструктури реагувати на зміну інтенсивності трафіку [23,10]. У реаліях багатодоменного середовища, де провайдер паралельно обслуговує десятки тисяч незалежних вебресурсів [1], архітектура мусить балансувати між двома полярними вимогами: максимізацією щільності розміщення без надлишкового резервування та жорсткими гарантіями SLA щодо доступності й часу відповіді [15,16]. Оскільки нерівномірність навантаження та взаємовплив орендарів є неминучими супутниками спільного використання обладнання [10,9], обрана модель має безперебійно підтримувати оперативний перерозподіл потужностей для згладжування локальних піків. Щоб обґрунтувати вибір цільової архітектури, нижче наведено критичний аналіз основних класів наявних рішень з акцентом на їхні сильні сторони та межі застосовності у масштабах масової експлуатації.

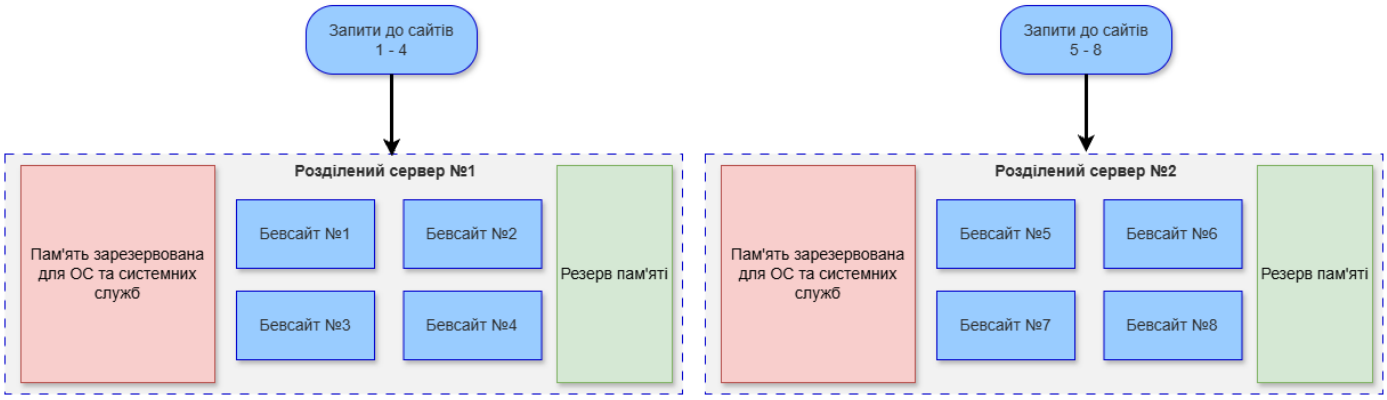

Рисунок 1.5 - Структурна схема розміщення вебресурсів в архітектурі класичного віртуального хостингу.

Історично найвищу щільність консолідації забезпечує класична модель спільного віртуального хостингу. У межах такої архітектури велика кількість вебсайтів функціонує в одному екземплярі операційної системи [1,24]. Це зменшує додаткові інфраструктурні витрати, оскільки не потребує створення окремого системного середовища для кожного

орендаря. Саме тому модель є економічно привабливою для масового розміщення слабо навантажених ресурсів [1].

Водночас її принциповим обмеженням є недостатня гнучкість керування ресурсами. Засоби обмеження та квотування на рівні операційної системи мають обмежену здатність гарантувати стабільну продуктивність у разі короткочасних пікових навантажень на окремі сайти [24,25,13]. Спільне використання процесорного часу, оперативної пам’яті та підсистеми введення-виведення призводить до конкуренції за ресурси. У разі навіть невеликих сплесків трафіку на окремий сайт погіршується якість обслуговування інших вебсайтів, що розміщені на тому самому фізичному вузлі. Подібні процеси деградації мають досить високу складність прогнозування та оперативного контролю [13].

Додатковим недоліком є висока інерційність міграції вебсайтів між фізичними вузлами. У традиційному віртуальному хостингу перенесення ресурсу зазвичай пов’язане з підготовкою оточення, перенесенням даних, синхронізацією стану та зміною налаштувань у конфігураціях сервісів. Такі операції, особливо налаштування DNS, потребують часу і мають обмеження щодо частоти виконання. Усі ці чинники ускладнюють оперативний перерозподіл навантаження та фактично унеможливлюють гнучку реакцію на зміну інтенсивності трафіку в режимі реального часу [23].

Архітектура, що ґрунтується на гіпервізорній віртуалізації та передбачає використання індивідуальних віртуальних серверів (VPS), орієнтована насамперед на ізоляцію клієнтського середовища (рис 1.6) [24,36].

У такій моделі гіпервізор забезпечує відокремлення ресурсів і контроль доступу до них. Це зменшує ризики взаємовпливу між орендарями та підвищує передбачуваність поведінки кожного ізольованого середовища за змінного навантаження. Унаслідок цього погіршення продуктивності одного клієнта не повинно безпосередньо впливати на інші віртуальні сервери в межах того самого фізичного вузла.

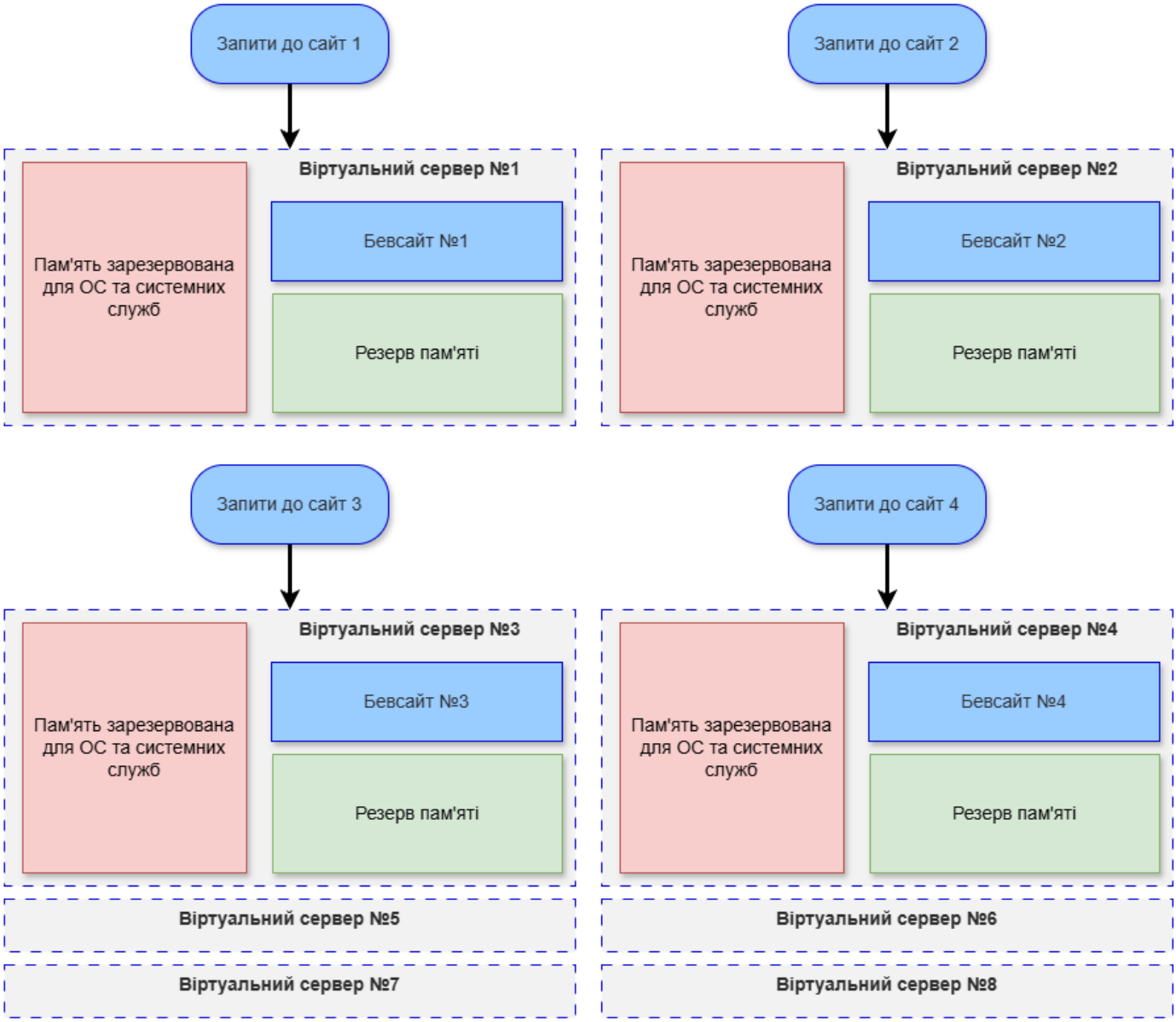

Рисунок 1.6 - Структурна схема розміщення вебресурсів в архітектурі віртуальних виділених серверів (VPS).

Разом із тим для масового сегмента слабо навантажених доменів така модель має суттєві обмеження економічного характеру. Кожна віртуальна машина потребує власного екземпляра гостьової операційної системи та базового набору системних служб [24,36]. Це зумовлює постійне фонове споживання оперативної пам’яті та інших ресурсів, незалежно від того, чи присутнє фактичне прикладне навантаження. Вказане явище доцільно трактувати як накладні витрати віртуалізації, тобто неминучий ресурсний запас, який витрачається на підтримання ізольованого системного

середовища. Такий запас є структурною складовою моделі й не може бути усунений без зміни підходу до ізоляції [37,36].

За умов, коли йдеться про десятки тисяч дрібних сайтів із низькою середньою інтенсивністю запитів, сума цих накладних витрат стає визначальною [1,2,38]. Значна частина оперативної пам’яті резервується для підтримання множини гостьових систем, а не для безпосереднього оброблення запитів [38]. Унаслідок цього знижується щільність розміщення вебресурсів на фізичних вузлах і зростає частка невикористаного потенціалу [8,38]. Отже, застосування моделі індивідуальних віртуальних серверів для великої кількості слабо навантажених доменів суттєво ускладнює ефективну консолідацію ресурсів і практично обмежує можливість економічно обґрунтованого масштабування в цьому сегменті [8,1,23].

Контейнерна віртуалізація на рівні операційної системи може розглядатися як проміжний підхід між класичним спільним хостингом і апаратною віртуалізацією (рис 1.7) [37,39].

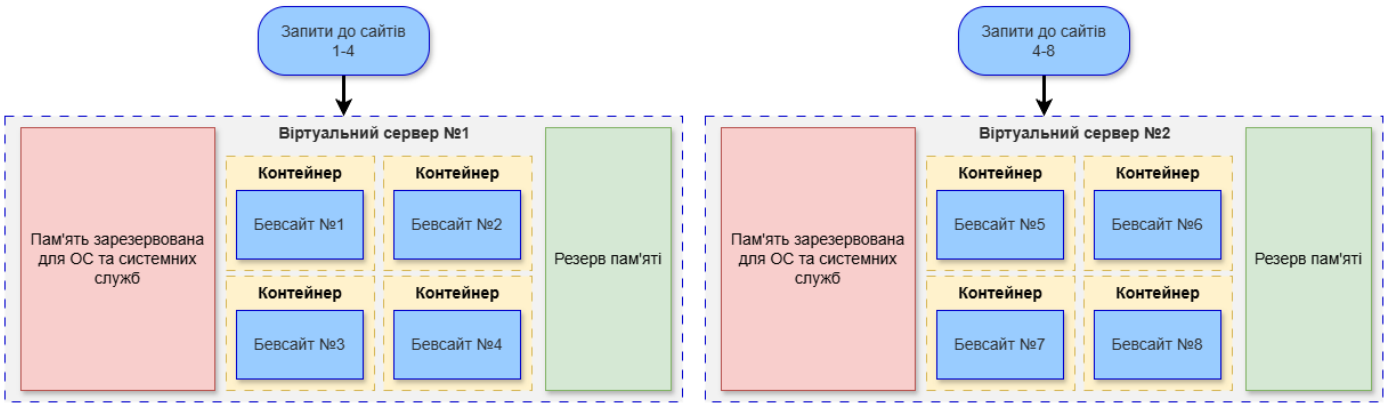

Рисунок 1.7 - Структурна схема розміщення вебресурсів в архітектурі контейнерізації.

Її основна ідея полягає в тому, що ізоляція досягається без запуску окремої гостьової операційної системи для кожного орендаря. Контейнери відокремлюють процеси за допомогою механізмів просторів імен та контрольних груп, що забезпечує розмежування доступу до ресурсів і встановлення обмежень на їх споживання. Завдяки цьому накладні витрати істотно менші, ніж у випадку повноцінних віртуальних машин,

що робить підхід привабливим для сценаріїв із великою кількістю відносно малих навантажень [38].

Разом із тим у масовому багатодоменному середовищі проявляються технологічні межі такого підходу. Першою з них є те, що виділення окремого контейнера для кожного слабо навантаженого сайту зумовлює дублювання середовища виконання в оперативній пам’яті [37,39]. Навіть якщо кожен сайт генерує незначний трафік, для кожного контейнера необхідно підтримувати робочі процеси, бібліотеки, кеші та інші структури, пов’язані з обраною технологічною платформою [39]. На практиці це означає повторне резервування пам’яті, наприклад для пулів інтерпретатора, модулів, конфігурацій та службових компонентів. У підсумку сумарне фонове споживання пам’яті зростає пропорційно кількості контейнерів і може обмежувати досяжну щільність консолідації [38].

Другою межею є операційна складність керування великою кількістю контейнерів [40,23]. За масштабів у десятки тисяч екземплярів зростають вимоги до систем розгортання, оновлення та інвентаризації, ускладнюється контроль версій, політик доступу й конфігурацій. Окремою проблемою стає маршрутизація мережевого трафіку до відповідного контейнера, підтримання правил балансування та забезпечення коректної роботи мережевих залежностей [41,42,43]. Також істотно збільшується навантаження на підсистеми моніторингу та журналювання, оскільки необхідно збирати та обробляти телеметрію для великої кількості ізольованих середовищ [23]. У сукупності зазначені чинники формують значні накладні витрати керування, що знижують практичну ефективність контейнерного підходу в сегменті масового розміщення слабо навантажених вебресурсів [44,23].

Керовані платформи оркестрації контейнерів розширюють можливості контейнерного підходу [45]. Вони забезпечують автоматизоване розгортання, декларативне керування бажаним станом системи та підтримують горизонтальне масштабування компонентів [46,47]. Для мікросервісних застосунків, де потрібні

складні взаємозв’язки між сервісами, контроль версій та стійкість до відмов, такий інструментарій є обґрунтованим і часто визначальним [41,42].

Однак у завданнях масового хостингу слабо навантажених вебресурсів виникає інша картина. Життєвий цикл більшості доменів відносно простий і не потребує розгалуженої внутрішньої топології чи складних сценаріїв розгортання. За таких умов починає домінувати базова вартість керування, пов’язана з підтриманням самого оркестраційного контуру. Додаткові компоненти, що забезпечують планування та оброблення подій, потребують значних ресурсів, що може суттєво зменшувати економічний ефект від консолідації малих навантажень [23,45,48].

Додатковим обмеженням є відповідність стандартних механізмів маршрутизації вимогам масового хостингу [41,42]. Для багатодоменного середовища критичним є ефективне термінування захищених з’єднань і масштабне керування криптографічними сертифікатами [49,50]. У практичних сценаріях провайдер має підтримувати десятки тисяч сертифікатів і регулярно оновлювати конфігурації маршрутів та політик доступу [49,51]. Типові реалізації маршрутизації в оркестраційних платформах проєктувалися для сервісної топології й не завжди оптимально пристосовані до високощільного термінування великої кількості сертифікатів [42,52]. Це створює технічні ризики, зокрема через каскадні перезавантаження контролерів під час зміни налаштувань, що ускладнює побудову масового хостингу на базі повнофункціональної оркестрації [43].

Архітектура безсерверних обчислень, у якій прикладна логіка реалізовується у вигляді подійно-орієнтованих функцій, забезпечує автоматичне масштабування відповідно до фактичного надходження запитів [53,54]. У такій моделі провайдер зосереджується на публікації функцій та їхніх залежностей, тоді як платформа бере на себе створення екземплярів виконання, розподіл навантаження та підтримання необхідної кількості паралельних обробників [53]. Це суттєво зменшує обсяг операційного керування інфраструктурою та усуває потребу вручну планувати ресурсний резерв для коливань трафіку.

Водночас для задачі вебхостингу, зокрема для систем керування контентом (CMS), застосовність такого підходу обмежена [55]. Першою причиною є ініціалізаційна затримка під час створення нового середовища виконання. У випадках, коли функція викликається після перерви або коли платформа повинна швидко збільшити кількість екземплярів, виникає додатковий час на підготовку середовища, завантаження залежностей та запуск виконання. Ця затримка призводить до помітного збільшення часу відповіді для частини запитів і критична для користувацьких сценаріїв, у яких очікується стабільна швидкодія [55,17,22].

Другою причиною є складність підтримання локального стану, який типово використовується у вебзастосунках на базі CMS [53,54]. До такого стану належать сесії користувачів, кеші застосунків, тимчасові дані виконання та інші структури, що покращують швидкодію за повторних звернень. Безсерверна модель, як правило, орієнтована на короткочасні виконання з мінімальною залежністю від локального середовища, а тому потребує винесення стану до зовнішніх сховищ. Це ускладнює архітектуру застосунку та збільшує кількість зовнішніх звернень, що може негативно впливати на затримки та на загальну вартість виконання [54,56,17].

Окремим обмежувальним чинником є економіка споживання ресурсів [53,54]. Модель тарифікації за кожен виклик функції зручна для нерегулярних коротких навантажень, коли оплата відповідає фактичному часу виконання [56]. Проте за тривалих пікових режимів або за стабільно підвищеної інтенсивності запитів сума платежів за численні виклики може перевищувати вартість використання виділених ресурсів у кластерній моделі [8,56]. У такій ситуації кластерні рішення, що дають змогу керувати обсягом резерву та ефективно розподіляти ресурси між багатьма доменами, можуть забезпечувати кращу економічну доцільність порівняно з помодульною оплатою за виконання функцій [8,23].

Порівняльний аналіз наведений у табл. 1.2 показує, що для масового хостингу слабо навантажених вебсайтів наявні на ринку універсальні архітектурні підходи не забезпечують одночасного досягнення всіх ключових вимог [8,1,23]. З одного боку,

рішення, орієнтовані на максимальну щільність розміщення, мають обмежені можливості керування ресурсами та не гарантують передбачуваної якості обслуговування в умовах локальних пікових режимів [57,33]. З іншого боку, підходи, що забезпечують високу ізоляцію та гнучке масштабування, супроводжуються зростанням накладних витрат і потребують значних ресурсів на підтримання середовища виконання та керувальних механізмів [36,38,55]. У підсумку жоден із універсальних варіантів не поєднує в межах однієї моделі максимально можливу щільність консолідації, оперативний і керований перерозподіл ресурсів та прийнятну обчислювальну й операційну складність у масштабі десятків тисяч доменів [8,1,23].

Таблиця 1.2 - Порівняльний аналіз архітектурних рішень для веб-хостингу.

Критерій порівняння	**Спільний хостинг**	**Віртуальні машини**	**Контейнеризація**	**Безсерверні**
Щільність консолідації	Найвища (статична)	Низька	Висока	Висока (динамічна)
Ступінь ізоляції	Низька	Найвища	Висока	Висока
Накладні витрати	Мінімальні	Високі (дублювання ОС)	Середні (дублювання Runtime)	Мінімальні (для користувача)
Швидкість масштабування	Низька (інерційна)	Низька (вертикальне)	Висока	Найвища
Масштабування TLS	Високе (статична)	Обмежене	Проблемне (перезавантаження)	Обмежене (ліміти платформ)
Робота зі станом	Природна	Природна	Природна	Ускладнена (зовнішня)

Зважаючи на виявлені обмеження, у дисертаційному дослідженні як цільовий підхід розглядається спеціалізована кластерна архітектура динамічного хостингу (рис. 1.8) [8,23]. Її призначенням є підтримання масового багатодоменного середовища за умов низької середньої інтенсивності трафіку та нерівномірних пікових режимів [1,4,9]. Оптимізаційний ефект у такій системі досягається не шляхом ізоляції кожного домену на рівні операційної системи [10]. Натомість він забезпечується через організацію спільного ресурсу кластера та керованого розподілу навантаження на рівні вхідної точки інфраструктури [8,58].

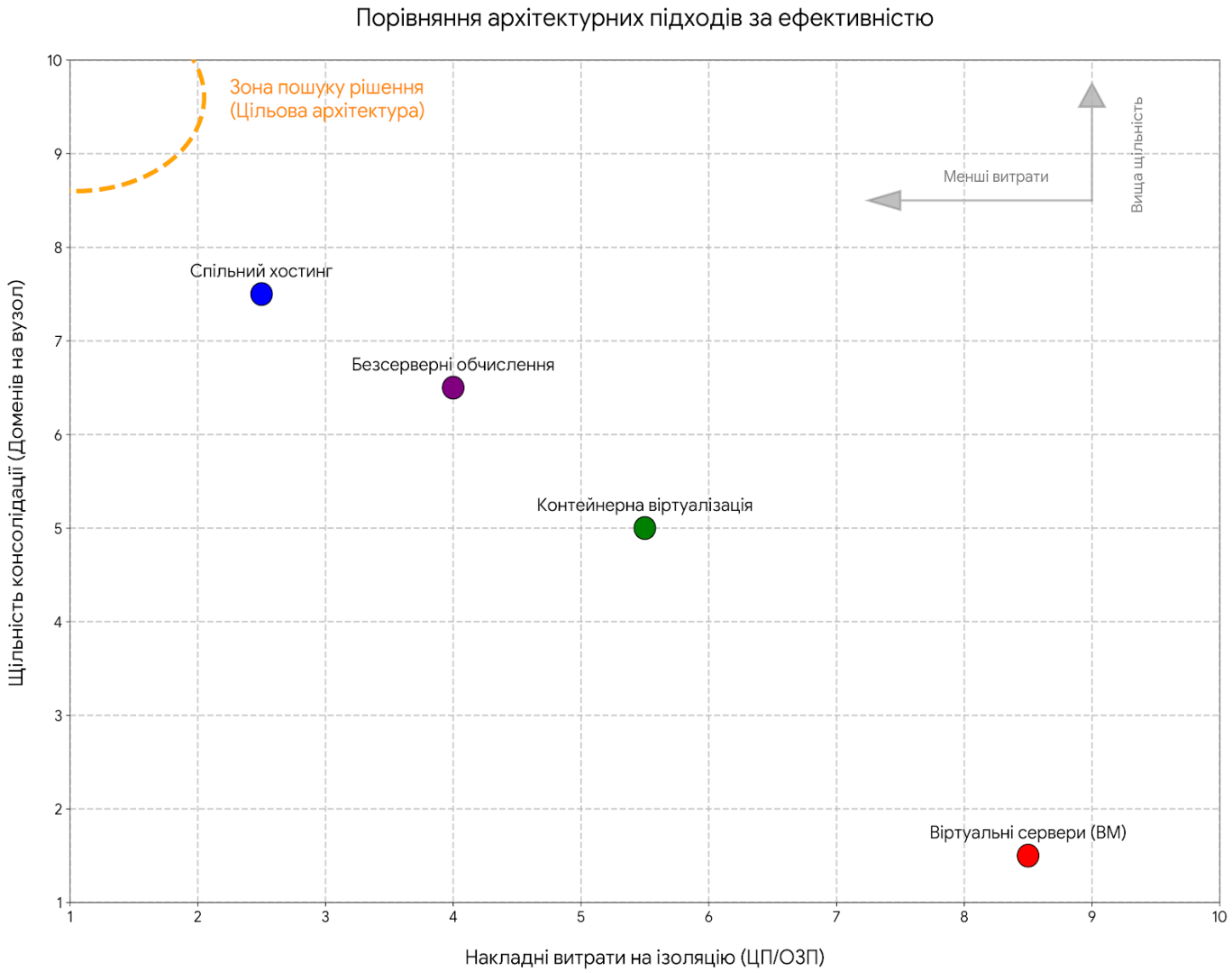

Рисунок 1.8 - Порівняльний аналіз ефективності архітектурних рішень за критеріями щільності консолідації та накладних витрат на ізоляцію.

Ключовим елементом запропонованого підходу є інтелектуальна точка входу, реалізована у вигляді балансувальника прикладного рівня [58,35]. Саме цей компонент виконує функції приймання та первинного оброблення запитів, приймає рішення щодо їх маршрутизації та забезпечує кероване використання спільних ресурсів кластера [58]. Важливою складовою архітектури є централізоване керування криптографічними протоколами [21]. Воно передбачає узгоджене термінування захищених з’єднань і централізовану роботу з криптографічними сертифікатами в умовах великої кількості доменів та регулярних змін конфігурацій [49,51]. Така організація спрощує експлуатаційні процедури, зменшує дублювання криптографічних налаштувань на

робочих вузлах і стабілізує часові характеристики встановлення з’єднань на вході системи [49,21].

Додатково оптимізація забезпечується застосуванням математичних алгоритмів планування ємності та керування резервом ресурсів [8,23]. Їхнє призначення полягає у визначенні необхідної кількості обчислювальних потужностей для виконання показників якості обслуговування за змінного трафіку [23,16]. Такі алгоритми мають враховувати стохастичну природу вхідних потоків, нерівномірність навантаження між доменами та можливість короткочасних пікових режимів [4,9]. У підсумку це дозволяє мінімізувати обсяг резервних потужностей без погіршення показників SLA [16]. Формалізацію моделей трафіку, критеріїв оптимізації та відповідних алгоритмічних процедур подається в наступних підрозділах.

1.3. Підходи до оптимізації використання обчислювальних ресурсів у хмарі

Оптимізація використання обчислювальних ресурсів у хмарній інфраструктурі є комплексним багатокритеріальним завданням [26]. Вона передбачає узгоджене застосування архітектурних рішень, механізмів керування та формалізованих математичних моделей [23]. На архітектурному рівні визначаються спосіб розміщення сервісів, принципи їх ізоляції та правила спільного використання ресурсів [10]. На рівні керування реалізовуються процедури, що змінюють доступну ємність кластера відповідно до поточного стану системи [23]. У попередньому оглядовому дослідженні автора з цієї тематики було систематизовано базові підходи до балансування навантаження в хмарній інфраструктурі та виокремлено ключові критерії їх ефективності, зокрема пропускну здатність, адаптивність, відмовостійкість, час відгуку і рівень використання ресурсів [59]. Математичні моделі забезпечують кількісне обґрунтування таких рішень. Вони дають змогу пов’язати характеристики трафіку, ресурсні обмеження та показники якості обслуговування в єдину формалізовану постановку [8].

Для надавача послуг масового хостингу це завдання має виражену специфіку. Оптимізація здійснюється не для окремого сервісу з прогнозованим профілем навантаження, а для сукупності з десятків тисяч доменів [1]. Для більшості з них індивідуальна інтенсивність звернень низька [1,5]. Однак сумарний потік запитів і число одночасних з'єднань формують істотні вимоги до стійкості інфраструктури, до передбачуваності часу відповіді та до виконання показників доступності [15,16]. Додатковою особливістю є нерівномірність розподілу навантаження між доменами та наявність короткочасних пікових режимів, які можуть суттєво впливати на споживання ресурсів [4,9]. Отже, економічна доцільність і виконання показників якості в такому середовищі визначаються здатністю інфраструктури ефективно консолідувати велику кількість малих навантажень і водночас керовано реагувати на відхилення від типових режимів роботи [8,23].

Одним із базових підходів до виконання цього завдання є консолідація. Під консолідацією в цьому контексті розуміється об'єднання незалежних стохастичних навантажень у спільний пул обчислювальних ресурсів, який використовується багатьма доменами. Такий підхід дає змогу перейти від ізольованого резервування потужностей для кожного вузла до керованого використання спільного резерву на рівні кластера [60,26]. У масовому хостингу це принципово, оскільки більшість сайтів мають низьку інтенсивність звернень, а отже ефективність визначається здатністю інфраструктури обслуговувати велику кількість малих потоків у межах обмеженого ресурсу [1,2,5].

Економічний і технічний ефект консолідації спирається на принцип статистичного мультиплексування [26]. Імовірність того, що пікові навантаження одночасно виникнуть для великої кількості незалежних доменів, мала [4,9]. Це означає, що спільний пул може підтримувати заданий рівень якості обслуговування, маючи менший сумарний резерв потужності, ніж потребувала б сукупність окремих вузлів у разі статичного підходу (рис. 1.9) [60,26].



Рисунок 1.9 - Моделювання процесу статистичного мультиплексування незалежних стохастичних навантажень.

Інакше кажучи, резерв, необхідний для забезпечення прийнятної роботи кластера, виявляється меншим за суму резервів, які довелося б тримати для кожного вузла за відсутності консолідації. Саме ця різниця і формує основний ефект підвищення ресурсної ефективності. На практиці зазначений ефект проявляється у зменшенні частки непродуктивного резервування ресурсів [60]. Для масового хостингу найбільш відчутним є зменшення резервування оперативної пам’яті, оскільки її обсяг безпосередньо обмежує кількість одночасно активних процесів та з’єднань [24]. Унаслідок переходу до спільного пулу знижується обсяг пам’яті, що простоює в очікуванні рідкісних пікових режимів, і зростає частка ресурсу, яка фактично використовується для оброблення запитів [60]. Це підвищує коефіцієнт корисного використання інфраструктури та створює передумови для економічно доцільного масштабування у сегменті багатодоменного хостингу [8].

Водночас консолідація не є автоматичним розв’язанням проблеми. Щоб її ефект був досяжним і стабільним, потрібні правила, які обмежують неконтрольоване споживання ресурсу окремими орендарями [13,10]. Такі правила мають задавати політики розміщення і допустимі норми ресурсного споживання в межах вузлів і кластера. Не менш важливими є механізми оперативного реагування на відхилення

трафіку від типових режимів [58,23]. До них належать процедури, що виявляють аномальну активність, локалізують її вплив та відновлюють рівновагу розподілу навантаження [35,28,33]. Без поєднання консолідації з керованими політиками та швидкою реакцією на аномалії підвищення щільності розміщення неминуче супроводжується зростанням ризику деградації якості обслуговування [23,10].

Другим фундаментальним напрямом виконання завдання оптимізації є динамічне масштабування обчислювальних ресурсів [27,23,61]. Його призначення полягає у зміні доступної ємності інфраструктури відповідно до поточного рівня навантаження [27,61]. Масштабування може здійснюватися у двох базових формах [61]. Горизонтальне масштабування полягає у зміні кількості активних обчислювальних вузлів, що беруть участь в обробленні запитів [47,61]. Вертикальне масштабування передбачає зміну обсягу ресурсу в межах одного вузла, зокрема збільшення або зменшення доступної оперативної пам’яті чи процесорної потужності [61,62]. Обидва підходи можуть застосовуватися окремо або у поєднанні, залежно від технологічних обмежень середовища та характеру навантаження [23,61].

У наукових джерелах і практиці експлуатації поширеним є розмежування масштабування за принципом прийняття керувальних рішень [23,63]. Реактивне масштабування ґрунтується на спостереженні апаратних і системних показників та використанні порогових правил [47,63]. Зазвичай до таких показників належать завантаження процесора, використання оперативної пам’яті, інтенсивність введення-виведення, довжина черг запитів та інші метрики, що безпосередньо відображають поточний стан вузлів [47]. За перевищення заданих порогів система ініціює додавання ресурсів, а за повернення метрик у допустимі межі може виконувати зменшення ємності [47]. Перевагою цього підходу є простота реалізації, однак його ефективність залежить від правильного вибору порогів і від швидкості реакції на зміну навантаження [47,63].

Проактивне масштабування орієнтується не стільки на порогові значення апаратних метрик, скільки на забезпечення цільового рівня якості обслуговування, визначеного угодами про рівень надання послуг (SLA) [63]. У цьому випадку керувальні

рішення базуються на прогнозуванні або оцінюванні майбутнього навантаження та на моделі зв’язку між параметрами системи, доступними ресурсами і показниками якості, зокрема часом відповіді та ймовірністю відмови в обслуговуванні [63]. Такий підхід дає змогу формувати резерв заздалегідь і зменшувати ризик деградації якості під час швидких сплесків трафіку [33,63]. Водночас він потребує формалізованих моделей, надійних оцінок параметрів і стабільних алгоритмів, здатних працювати в умовах стохастичної природи навантаження та багатодоменного середовища [8,4,10].

У системах масового вебхостингу використання усереднених показників стану вузлів часто не дає надійної основи для керування ресурсами [23,63]. Зокрема, орієнтація лише на середні значення завантаження центрального процесора недостатньо. Причина полягає в тому, що погіршення якості обслуговування в таких середовищах зазвичай зумовлюється не тривалим зростанням середнього навантаження, а короткочасними подіями [4,63]. До них належать мікросплески одночасних запитів, різке збільшення кількості активних з’єднань або швидке вичерпання оперативної пам’яті [8,4,63]. Такі явища мають імпульсний характер і можуть істотно впливати на час відповіді та ймовірність відмов навіть тоді, коли середні значення метрик залишаються в межах допустимого діапазону [17,63].

Додатково слід зважати на те, що усереднення метрик за фіксований інтервал часу приховує критичні короткі коливання [47,63]. Пікові значення, які фактично призводять до черг, відмов або примусового вивільнення пам’яті, можуть бути статистично згладжені під час агрегування показників за хвилину або за інший інтервал моніторингу [33,63]. Унаслідок цього система керування отримує запізнілу або недостатньо чутливу інформацію про наближення до дефіциту ресурсу [47,63]. Це підвищує ймовірність того, що масштабування буде ініційоване вже після появи помітної деградації якості обслуговування [47,63].

Отже, контур керування масштабуванням у масовому хостингу має спиратися на показники, які безпосередньо відображають стан дефіцитного ресурсу та чутливі до короткочасних перевантажень [61,63]. Для практичної реалізації це означає пріоритет

метрик, що характеризують доступність оперативної пам’яті, рівень конкурентності оброблення запитів, накопичення черг і частоту відмов, а також показників, які мають прямий зв’язок із критеріями якості обслуговування [23,16,47,62,63]. Такий вибір метрик дає можливість будувати керування, орієнтоване на фактичні ризики деградації, а не лише на усереднену інтенсивність навантаження.

Процес масштабування тісно пов’язаний із плануванням ємності [8,63]. Масштабування забезпечує оперативне коригування стану системи в режимі близькому до реального часу [61,63]. Воно реагує на зміну інтенсивності трафіку та поточне споживання ресурсів, додаючи або вивільняючи потужності відповідно до керувальних правил [61,63]. Натомість планування ємності виконує іншу функцію. Воно задає базову конфігурацію кластера та визначає мінімально необхідний обсяг ресурсів, який має бути доступним у стабільному режимі експлуатації [8].

У межах планування ємності встановлюється мінімальна кількість обчислювальних вузлів, достатня для поглинання стохастичних сплесків навантаження без переходу системи в режим дефіциту ресурсу [8,4]. Водночас береться до уваги необхідність дотримання заданих показників якості обслуговування, що формалізуються через параметри SLA [8,16]. Отже, планування ємності формує нижню межу ресурсного забезпечення, на яку спираються алгоритми масштабування під час короткочасних коливань трафіку [8,63].

На довгостроковому горизонті планування ємності виконує роль інструмента оптимізації економіки інфраструктури [23,60]. Воно забезпечує баланс між вартістю утримання ресурсів і ризиком порушення показників SLA [23,16]. Надмірне резервування підвищує витрати на інфраструктуру та знижує ефективність консолідації [8,60]. Недостатнє резервування, навпаки, збільшує ймовірність деградації продуктивності та відмов у пікових режимах [23,63]. Отже, коректне планування ємності є необхідною умовою для того, щоб масштабування працювало в межах обґрунтованої базової конфігурації та забезпечувало стійкі показники якості обслуговування у багатодоменному середовищі [8,23,63].

У традиційних схемах хостингу провайдери широко застосовують практику надлишкового виділення ресурсів [64]. Її зміст полягає у тому, що сумарні ліміти, призначені клієнтам, свідомо перевищують фізично доступну ємність сервера [64]. Такий підхід поширений, оскільки значна частина орендарів не використовує заявлені ліміти одночасно [26]. Для окремих типів ресурсів надлишкове виділення може бути прийнятним. Зокрема, для процесорного часу та мережевої пропускної здатності ризики відносно керовані, оскільки операційна система використовує механізми розподілу процесорного часу між процесами та згладжує короткочасні пікові навантаження за рахунок черг і планувальника [24]. У таких умовах перевищення сумарних лімітів не обов’язково призводить до миттєвої відмови, хоча й може збільшувати затримки [24].

Для оперативної пам’яті ситуація принципово інша [24]. Пам’ять є ресурсом із жорсткою фізичною межею, а перевищення доступного обсягу має прямі та швидкі наслідки для працездатності [24]. Вичерпання фізичної пам’яті призводить або до інтенсивного використання підкачки, що різко збільшує затримки оброблення запитів і викликає деградацію продуктивності (рис. 1.10), або до активації механізмів примусового завершення процесів для відновлення доступного обсягу пам’яті [24,25]. У типовому випадку це реалізовується через механізм аварійного завершення процесів у разі дефіциту пам’яті, що супроводжується зупинкою частини робочих компонентів вебсередовища та втратою обслуговування запитів [57,24,25]. За багатодоменного розміщення такі події мають системний характер [10]. Вони здатні спричиняти каскадні відмови, коли зростання затримок та повторні спроби клієнтів додатково підсилюють навантаження на інфраструктуру [65].

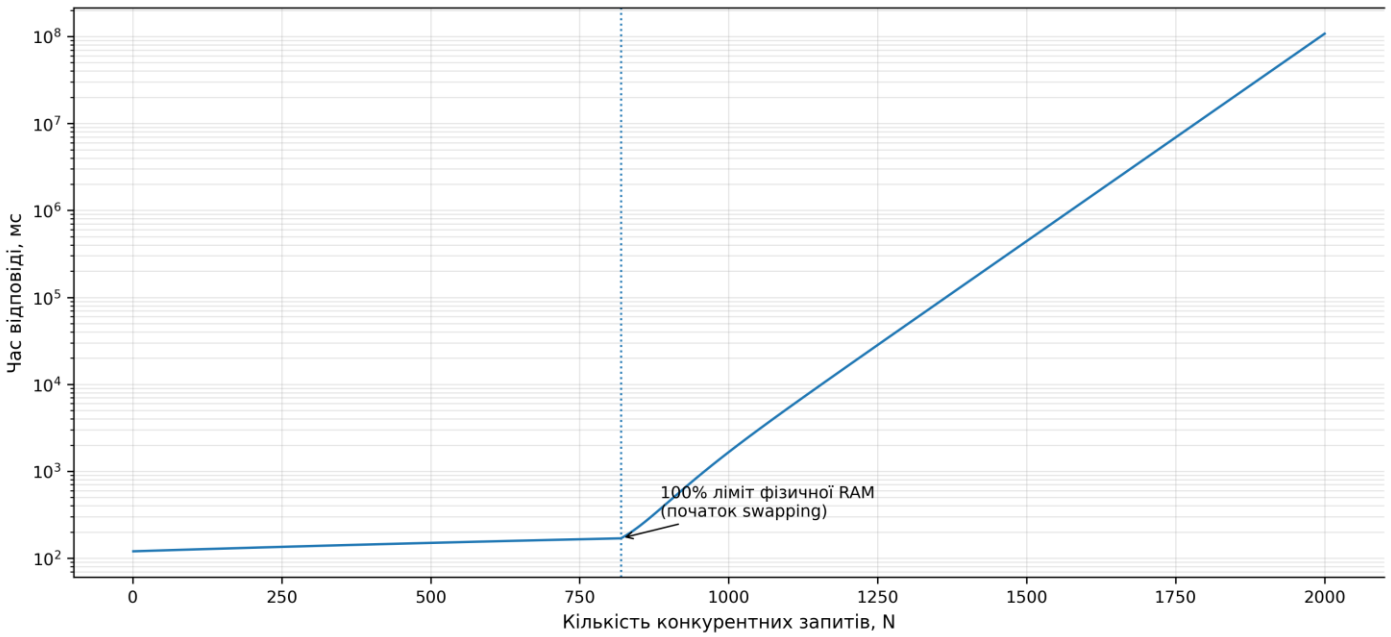

Рисунок 1.10 - Модель впливу надлишкового виділення пам’яті на час відповіді: експоненційне зростання затримки після досягнення ліміту фізичної RAM.

З огляду на це, базовим об’єктом оптимізації стає не лише середня інтенсивність трафіку, а насамперед рівень конкурентності оброблення запитів [8,63]. Рівень конкурентності визначається кількістю одночасно активних процесів, потоків або з’єднань, які обслуговуються в певний момент часу [24]. Саме цей показник прямо пов’язаний зі споживанням оперативної пам’яті, оскільки кожне активне обслуговування потребує виділення пам’яті під контекст виконання та супутні структури [8,24]. Тому керування конкурентністю є більш адекватним інструментом забезпечення стійкості системи, ніж контроль лише усереднених показників навантаження [23,63].

У кластерній моделі принцип організації резерву змінюється [66,26]. Резерв потужності формується не як статично зафіксована квота для кожного окремого вузла. Натомість він розглядається як спільний керований запас у межах загального пулу ресурсів [66,23,26]. Така організація допомагає перерозподіляти доступну ємність між вузлами відповідно до фактичної конкурентності та поточного стану пам’яті [8,23]. Як наслідок, змінюється економіка обслуговування: зменшується потреба утримувати

надлишкові ресурси на кожному сервері та зростає ефективність використання апаратної бази за умови контролю ризиків порушення SLA [8,23,16].

Зазначені фізичні обмеження зумовлюють потребу в математичному апараті, який дає змогу узгоджено описати три взаємопов'язані аспекти роботи інфраструктури [8,23,63]. Першим є випадковий характер надходження вебзапитів, що проявляється у нерівномірності інтенсивності трафіку та наявності короткочасних пікових режимів [4,9,63]. Другим є жорстка обмеженість обчислювальних ресурсів, насамперед оперативної пам'яті, яка визначає допустиму конкурентність оброблення [8,24,25]. Третім є ймовірнісні вимоги до якості обслуговування, що фіксуються в угодах про рівень надання послуг і можуть бути виражені через допустиму частку відмов, граничні затримки та цільову доступність [8,15,23,16].

Оскільки вхідний вебтрафік є випадковим процесом, його опис лише детермінованими величинами недостатній [4]. Для побудови моделей, що відображають імовірність перевантаження, втрати запитів або порушення часових обмежень, доцільно застосовувати методи теорії масового обслуговування [14]. Апарат систем масового обслуговування надає інструменти для встановлення зв'язку між інтенсивністю надходження запитів, характеристиками їх обслуговування, доступною ємністю системи та показниками якості [14]. Це створює формальну основу для подальшого обґрунтування правил планування ємності, вибору резерву та побудови алгоритмів керування ресурсами у кластерній інфраструктурі [8,23].

Для завдання динамічного вебхостингу особливе значення мають моделі систем із відмовами, тобто втратні моделі [67,68]. Їх застосування обґрунтоване у ситуаціях, коли ресурс, що обмежує роботу системи, має жорстку фізичну межу і не допускає накопичення необмеженої черги [67,68]. У випадку масового хостингу таким ресурсом виступає оперативна пам'ять [8,25]. Саме вона лімітує кількість процесів або робочих контекстів, які можуть одночасно обробляти запити [8,24,25]. Отже, число паралельних обслуговувань можна інтерпретувати як кількість каналів обслуговування [8,67].

За повного використання доступних каналів новий запит не може бути довільно відкладений на очікування [14]. Причина полягає в тому, що його приймання також потребує пам’яті та робочих ресурсів, а збільшення черги в таких умовах лише поглиблює дефіцит і підсилює деградацію якості [65,14]. Тому в практичній реалізації запити, що надходять у момент, коли всі канали зайняті, відхиляються на вході системи [14]. Це може виконуватися балансувальником прикладного рівня або іншим компонентом точки входу, який приймає рішення про відмову в обслуговуванні з метою збереження працездатності системи та недопущення лавиноподібного перевантаження [58,14]. У межах моделі така ситуація формалізується як втрата запиту через відсутність вільного каналу [14].

У цьому контексті доцільним математичним інструментом є ерлангівські втратні моделі [14,67,68]. Зокрема, формула Ерланга B дає можливість кількісно оцінювати ймовірність втрати запиту залежно від запропонованого навантаження та кількості каналів обслуговування [14,67,68]. Запропоноване навантаження визначається інтенсивністю надходження запитів і середнім часом їх оброблення [14,67]. Кількість каналів відображає доступну конкурентність, яка у цьому завданні обмежується обсягом оперативної пам’яті [8,25,14]. Отже, формула Ерланга B встановлює нелінійний зв’язок між інтенсивністю вхідного трафіку, середньою тривалістю оброблення та необхідною кількістю каналів для забезпечення заданого рівня якості, який може бути виражений через цільову ймовірність втрати запиту [8,14,68]. Це створює основу для подальшого планування ємності та вибору мінімально достатнього ресурсу за заданих вимог SLA [8,23,16,14].

Застосування формули Ерланга B дає змогу інтерпретувати обчислювальний ресурс не через узагальнені показники на кшталт відсотка завантаження процесора, а через конкретну місткість системи, що визначається доступним обсягом оперативної пам’яті [8,25,14]. У такій постановці ресурс подається як дискретна кількість одночасно підтримуваних обслуговувань, кожне з яких потребує певного фіксованого обсягу пам’яті для створення та підтримання робочого контексту запиту [8,25]. Відповідно,

поняття «канал обслуговування» набуває прикладного змісту [8,14]. Воно відображає максимально можливу конкурентність, яку може забезпечити вузол або кластер без переходу в режим дефіциту пам’яті [8,25].

За цих умов завдання оптимізації ресурсоспоживання формулюється як завдання вибору мінімально достатньої ємності системи [8,14,68]. Ідеться про визначення такої кількості каналів обслуговування, яка забезпечує заданий рівень якості за стохастичного характеру вхідного трафіку [14,67,68]. Критерієм якості виступає ймовірність відмови в обслуговуванні запитів, що виникає у випадках, коли всі канали зайняті [14,67,68]. Використовуючи ерлангівську модель, можна аналітично пов’язати цю ймовірність з інтенсивністю надходження запитів, середнім часом їх оброблення та кількістю доступних каналів [14,67,68].

Отже, оптимізація зводиться до аналітичного визначення мінімальної ємності кластера, за якої ймовірність відмови не перевищує граничних значень, установлених угодами про рівень надання послуг (SLA) [8,16,14]. Така постановка дозволяє перейти від інтуїтивного резервування ресурсів до формалізованого обґрунтування базової конфігурації інфраструктури, а також створює основу для подальшого поєднання планування ємності з алгоритмами динамічного масштабування [8,63].

У попередній публікації автора, присвяченій оптимізації розподілу навантаження в інфраструктурі хмарного сервісу, було розглянуто окремі підходи до оцінювання ефективності керувальних дій та сформульовано відповідні критерії [58,59]. У межах цієї дисертації зазначені результати отримують подальший розвиток і уточнення. Акцент переноситься на оперативну пам’ять як на ресурс, що найчастіше визначає граничну конкурентність оброблення запитів у масовому вебхостингу [8,25]. Водночас стохастичний характер вхідного трафіку обґрунтовує застосування втратних моделей як інструмента кількісного опису ймовірності відмов та формування правил динамічного планування ємності [8,4,14].

Строгу математичну постановку задачі, обрані моделі та отримані залежності наведено в розділі 2. Їх експериментальну перевірку в характерних сценаріях

функціонування системи передбачено подати у розділі 4. Така логіка викладу забезпечує зв’язок між теоретичним обґрунтуванням і подальшою експериментальною верифікацією запропонованих підходів.

Отже, розв’язання задачі масового хостингу слабо навантажених вебресурсів потребує поєднання двох взаємодоповнювальних складових. З одного боку, необхідна інфраструктурна консолідація, що підвищує ефективність використання апаратної бази [60,26]. З іншого боку, потрібні математичні методи теорії масового обслуговування, які дають змогу формалізувати стохастичну природу трафіку та обґрунтувати параметри ємності й керувальних дій з урахуванням імовірнісних вимог SLA [8,4,16,14]. Саме їх узгоджена інтеграція формує методичну основу подальших розділів дослідження.

1.4. Балансування навантаження і керування TLS у багатодоменному середовищі

У кластерній моделі динамічного хостингу балансувальник прикладного рівня, що функціонує на сьомому рівні еталонної моделі OSI, виконує роль не лише засобу маршрутизації мережевих запитів [58,69]. Він також є центральним інтеграційним компонентом, через який узгоджуються вимоги до продуктивності, керованості та якості обслуговування [58,23]. Саме на рівні точки входу зосереджуються механізми, що визначають часові характеристики оброблення запитів і забезпечують контрольований доступ до ресурсів кластера [58,35,21,22].

На цьому рівні реалізовується термінування захищених з’єднань і виконуються операції, пов’язані з криптографічним встановленням сесії [52,49,21]. Далі здійснюється визначення цільового домену на основі параметра розширення ідентифікації імені сервера, що передається під час встановлення TLS-з’єднання [70]. Після ідентифікації домену балансувальник приймає рішення щодо маршрутизації запиту до відповідного середовища виконання, яке забезпечує логічну ізоляцію обслуговування [58,10]. Одночасно цей компонент виступає місцем концентрації експлуатаційних даних [58]. Він забезпечує збирання телеметрії щодо інтенсивності трафіку, характеристик з’єднань

і параметрів оброблення запитів, що надалі використовується для моніторингу, планування ємності та керування масштабуванням [8,58,63].

У багатодоменному середовищі, яке одночасно підтримує десятки тисяч незалежних вебсайтів, точка входу набуває визначального значення для масштабованості всієї системи. Саме її пропускна здатність, затримки оброблення та стійкість до перевантажень формують практичну верхню межу зростання кластера [58,21,22,65]. Якщо цей компонент стає «вузьким місцем», подальше нарощування обчислювальних вузлів не забезпечує очікуваного ефекту, оскільки вхідний тракт не здатний прийняти та коректно розподілити додатковий обсяг запитів [58,31,65].

Через точку входу проходить весь легітимний трафік, включно зі встановленням захищених з'єднань і маршрутизацією до відповідних середовищ виконання [58,49,21]. На цьому ж рівні проявляються аномальні режими роботи, зокрема різкі сплески кількості запитів та інші форми шкідливої активності, які впливають на споживання ресурсів і на показники якості обслуговування [28,33,31]. Окрім того, саме на рівні точки входу концентруються операції, пов'язані з керуванням середовища. Тут застосовуються динамічні зміни конфігурації кластера, оновлюються правила маршрутизації, актуалізуються криптографічні параметри та впроваджуються обмеження, спрямовані на збереження стабільності системи [49,43,51,65]. У сукупності це робить точку входу ключовим елементом, від якого залежить як продуктивність, так і керованість багатодоменної інфраструктури [58,23,10].

Визначальною ознакою сучасного масового вебхостингу є перевага трафіку, що передається із застосуванням шифрування [19,20]. За таких умов розподільник навантаження на вході системи вимушений обробляти HTTPS-з'єднання як основний тип взаємодії клієнта з вебресурсом [19,21]. Це змінює профіль витрат на оброблення запитів, оскільки до типових операцій маршрутизації додаються криптографічні процедури встановлення захищеної сесії [71,21].

Під час оброблення HTTPS-запитів балансувальник виконує операції криптографічного узгодження параметрів з'єднання, зокрема процедуру встановлення

TLS-сесії [21]. На цьому етапі застосовуються алгоритми асиметричної криптографії, які суттєво затратніші за процесорним часом, ніж подальше симетричне шифрування даних після встановлення ключів [71,21]. Саме початкова фаза встановлення сесії формує основну частину обчислювальних витрат на один новий зв'язок і визначає чутливість точки входу до короткочасних сплесків кількості з'єднань [71,21].

Окрім криптографічних обчислень, балансувальник має виконувати операцію вибору цифрового сертифіката для конкретного домену [49,70]. У багатодоменному середовищі кількість сертифікатів може становити десятки тисяч, а відповідний сертифікат необхідно обрати до завершення встановлення захищеного з'єднання [58,70]. Вибір здійснюється на основі доменного імені, переданого клієнтом під час ініціалізації сесії, і має виконуватися з мінімальною затримкою [21,22,70]. Отже, ефективність механізму пошуку сертифіката та організація відповідних структур даних на вході системи стають критичними для забезпечення продуктивності й масштабованості масового вебхостингу [49,21,22].

У масовому хостингу завдання керування сертифікатами ускладнюється тим, що їхній життєвий цикл безперервний [72]. Сертифікати автоматично випускаються, регулярно поновлюються та за потреби відкликаються. Кількість таких подій у багатодоменному середовищі значна, а отже зміни в наборі сертифікатів і пов'язаних з ними налаштуваннях мають відбуватися постійно [49]. У цих умовах система повинна працювати так, щоб операційні процедури керування сертифікатами не перетворювалися на причину простою або нестабільної роботи точки входу [43,51].

Тому однією з базових вимог до інфраструктури є підтримання динамічного застосування змін без зупинки сервісу [43,51]. Ідеться про можливість оновлювати конфігурацію та актуалізувати набір сертифікатів у робочому режимі [49]. Такі оновлення не повинні призводити до розриву активних сесій і не мають погіршувати часові характеристики оброблення запитів для легітимних користувачів [51,22]. Практично це означає, що процес внесення змін має бути відокремлений від критичного

шляху оброблення запиту та не створювати додаткових затримок у моменти застосування конфігурації [43,22].

З інженерного погляду виконання цієї вимоги потребує ефективної організації даних, що використовуються для вибору сертифіката та пов’язаного контексту. Пошук відповідного запису має виконуватися з мінімальною обчислювальною складністю та з прогнозованою затримкою навіть за наявності десятків тисяч доменів [49,70]. Для цього можуть застосовуватися структури даних, що забезпечують швидкий доступ за доменним іменем, зокрема префіксні дерева або оптимізовані хеш-таблиці [73]. Водночас критичним є зменшення блокувань під час доступу до спільних структур у пам’яті. Механізм оновлення має мінімізувати конкуренцію потоків і не створювати взаємного очікування в критичному шляху оброблення мережевого запиту [74]. Це є необхідною умовою для підтримання стабільної продуктивності точки входу за одночасної наявності інтенсивного трафіку та частих змін конфігурації [43,51].

Окрім функції термінування шифрованих з’єднань, балансувальник прикладного рівня виконує роль основного джерела первинних експлуатаційних показників, необхідних для моніторингу, керування трафіком та якістю обслуговування [58]. Саме на рівні точки входу формується узгоджене уявлення про поточний стан системи, оскільки через цей компонент проходять усі запити, а отже він має можливість безпосередньо спостерігати як параметри вхідного потоку, так і реакцію інфраструктури на навантаження [58]. Отримані показники використовуються для оцінювання дотримання цільових параметрів, визначених угодами про рівень надання послуг, та для ініціювання керувальних дій у межах масштабування і планування ємності [8,63].

До базових показників, що реєструються на рівні балансувальника, належить інтенсивність вхідного потоку запитів [58]. Вона відображає фактичну кількість звернень за одиницю часу та дає змогу фіксувати як довгострокові тенденції, так і короткочасні сплески активності. Не менш важливим є вимірювання тривалості оброблення запитів на обчислювальних вузлах [58]. Цей показник характеризує швидкодію середовища виконання, дає змогу оцінити вплив ресурсних обмежень і

використовується як вхідний параметр для моделей, що пов’язують навантаження з показниками якості [8,14].

Окрему групу становлять метрики, що описують рівень конкурентності. До них належать кількість активних з’єднань і кількість паралельних обслуговувань у певний момент часу [58]. Саме ці показники критичні для масового хостингу, оскільки вони тісно пов’язані зі споживанням оперативної пам’яті та з ризиком переходу системи в режим дефіциту ресурсу [25]. Додатково балансувальник може формувати непрямі індикатори деградації, які сигналізують про наближення до граничного режиму. До таких індикаторів належать, зокрема, зростання довжини черги запитів, збільшення частоти повторних спроб з боку клієнтів та накопичення затримок на етапі встановлення з’єднань [21,65]. Використання сукупності зазначених метрик дозволяє не лише фіксувати факт погіршення якості, а й своєчасно виявляти передумови деградації та керовано реагувати на них [63].

У парадигмі динамічного хостингу зібрані експлуатаційні дані мають активне призначення. Вони використовуються не лише для спостереження за станом системи, а й як основні вхідні сигнали для контуру зворотного зв’язку. У такому контурі вимірювання, оцінювання та керувальні дії утворюють єдину послідовність. Стан інфраструктури безперервно співвідноситься з вимогами до якості обслуговування, а відхилення від цільових параметрів мають приводити до керованого коригування доступної ємності [63].

Розподільник навантаження в цій архітектурі посідає центральне місце, оскільки є точкою, через яку проходить весь трафік [58]. Це робить його єдиним компонентом, здатним формувати об’єктивну оцінку фактичних параметрів стохастичного потоку запитів у реальному режимі. На відміну від окремих обчислювальних вузлів, які бачать лише свою частину навантаження, точка входу спостерігає сумарну картину та може коректно фіксувати зміни інтенсивності, рівень конкурентності та характер пікових відхилень [63].

На основі цих вимірювань балансувальник здійснює зіставлення поточних показників із цільовими значеннями, визначеними в SLA [16]. Таке зіставлення може виконуватися як за прямими метриками якості (наприклад, час відповіді або частка відмов), так і за показниками, що відображають наближення до дефіциту ресурсу. У разі виявлення відхилень або передумов деградації розподільник навантаження ініціює передавання керувальних сигналів до підсистеми масштабування. Це забезпечує замкнений характер керування ємністю кластера (рис. 1.11), у якому рішення про зміну ресурсу спираються на фактичні параметри трафіку та на вимоги до якості обслуговування, а не лише на усереднені апаратні показники окремих вузлів [63].

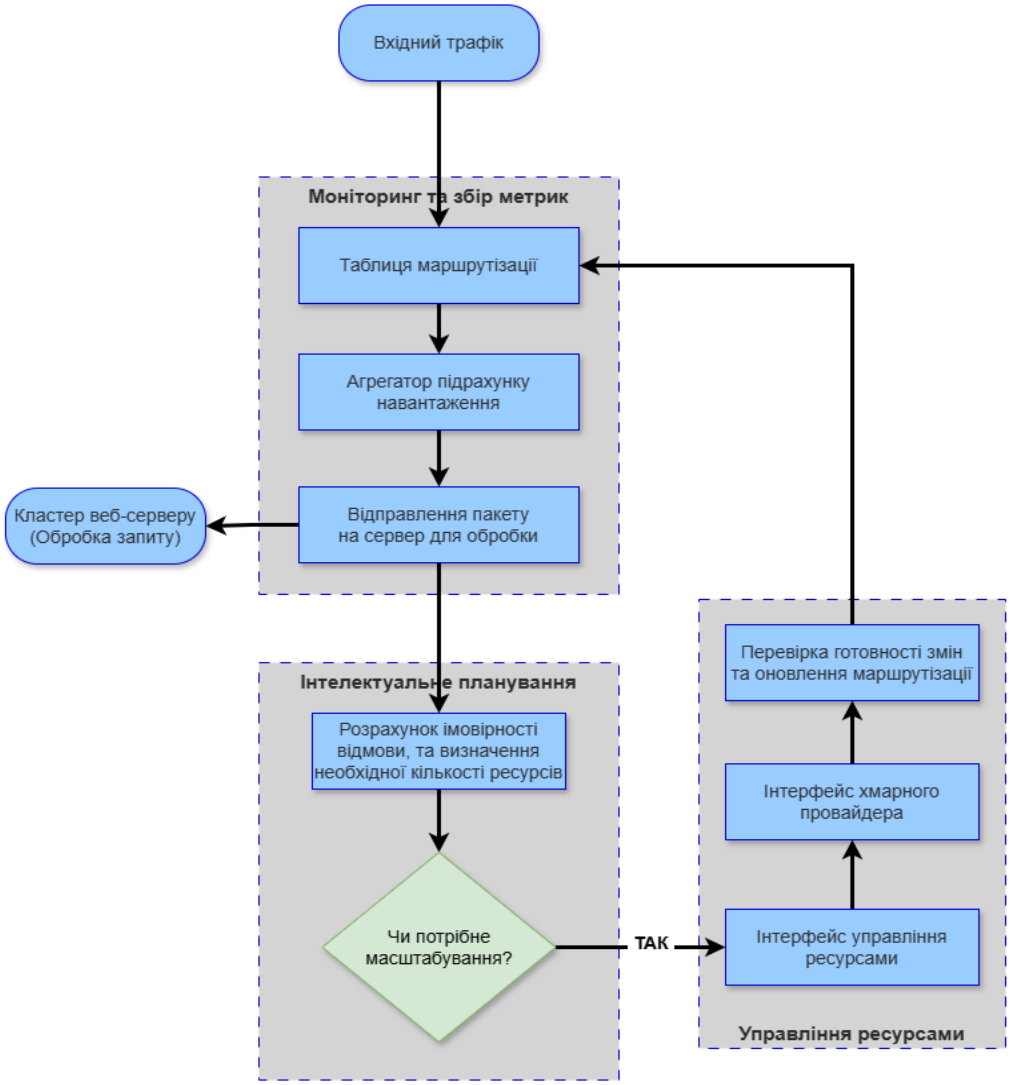

Рисунок 1.11 - Схема інформаційної технології динамічного масштабування на основі ймовірнісних характеристик СМО.

Алгоритми маршрутизації запитів, зокрема циклічний розподіл, вибір вузла з мінімальною кількістю активних з’єднань або хешування за параметрами джерела, у цій роботі розглядаються на рівні базових принципів. Такий вибір зумовлений тим, що розроблення нового методу маршрутизації не є основною метою дослідження. У межах дисертації маршрутизація розглядається як необхідний функціональний механізм точки входу, який забезпечує доставлення запитів до середовищ виконання та створює умови для подальшого керування ресурсами.

Для масового хостингу вирішальними є інші критерії, які визначають практичну придатність балансувальника в багатодоменному середовищі [58]. Насамперед ідеться про здатність безпечно та стабільно виконувати значні обсяги криптографічних операцій, що супроводжують встановлення захищених з’єднань, і робити це з передбачуваними затримками [71,21]. Не менш важливою є підтримка інтеграції з керувальними підсистемами, які працюють асинхронно та впливають на поведінку точки входу [35]. До таких підсистем належать механізми захисту від прикладних атак та контури масштабування ємності, що приймають рішення на основі телеметрії та оновлюють правила оброблення запитів у робочому режимі [35,63]. Отже, у цій задачі ключовим є не ускладнення алгоритму маршрутизації, а забезпечення керованості, стійкості та інтеграційної здатності точки входу за високої щільності доменів і динаміки конфігурацій [43].

Проблемна площина, що безпосередньо пов’язана з науковим внеском цієї роботи, полягає у невідповідності традиційних підходів до оцінювання продуктивності систем розподілу навантаження вимогам масового динамічного хостингу. Аналіз фахових джерел показує, що більшість еталонних тестувань ґрунтується на синтетичних сценаріях, у яких вимірюють граничну пропускну здатність і середню затримку за фіксованої конфігурації. У межах таких експериментів система розглядається як відносно незмінний об’єкт, а вплив керувальних дій і динамічних оновлень або ігнорується, або виноситься за межі методу [58,75,76].

Для реального середовища динамічного хостингу така методологія є недостатньою [75,76]. Визначальні чинники деградації якості виникають не лише через збільшення середнього трафіку, а через специфічні режими роботи точки входу. До них належать масове встановлення захищених з'єднань, безперервна актуалізація цифрових сертифікатів та оперативне застосування змін у правилах доступу й маршрутизації [43,21,72]. Саме ці сценарії формують додаткові витрати процесорного часу, створюють конкуренцію за ресурси в критичному шляху та можуть спричиняти нестабільність часових характеристик, навіть якщо середня інтенсивність запитів залишається помірною [43,21,74].

Отже, у зазначених умовах балансувальник функціонує не як статичний комутатор мережевих запитів. Він працює як компонент, що постійно перебуває під впливом керувальних подій [58]. Конфігурація, набір сертифікатів, правила оброблення трафіку та захисні політики змінюються в робочому режимі, а сам процес застосування змін стає частиною експлуатаційного навантаження [43,72,77]. Тому оцінювання продуктивності має зважати не лише на пропускну здатність у стаціонарних умовах, а й на поведінку системи під час частих оновлень, стійкість до керованих змін і здатність зберігати показники якості обслуговування за перманентної динаміки конфігурації [43,75,76].

У публікації автора, присвяченій методології вибору балансувальника на основі динамічного підходу, було виявлено розрив між типовими статичними оцінками та реальними експлуатаційними режимами [58]. Показано, що вимірювання лише граничної пропускної здатності та середньої затримки за незмінної конфігурації не відтворює умов, у яких фактично працює точка входу в багатодоменному середовищі [75]. У запропонованій методології акцент зроблено на оцінюванні стійкості характеристик у часі. Така стійкість розглядається як здатність системи зберігати прогнозовані показники продуктивності та затримок під впливом керувальних подій, зокрема під час застосування змін конфігурації, оновлення сертифікатів і модифікації правил оброблення трафіку [43,72].

У межах цієї дисертації зазначена методологія використовується як вихідна основа для формування вимог до інтелектуальної точки входу. Вона визначає, які саме експлуатаційні режими мають бути враховані під час проєктування балансувальника прикладного рівня та які властивості критичні для забезпечення якості обслуговування у динамічному середовищі. Окрім того, методологія слугує основою для планування експериментальних сценаріїв, які буде використано для оцінювання практичної придатності запропонованих рішень. Зокрема, експериментальна перевірка має показати, що запропонована архітектура здатна підтримувати задані показники якості обслуговування в умовах постійної динаміки конфігурації, високої інтенсивності встановлення захищених з'єднань та регулярного застосування керувальних змін. Таким чином, методологія динамічного оцінювання виступає не лише інструментом порівняння балансувальників, а й практичним підґрунтям для подальшої верифікації ключових положень дисертаційного дослідження.

Отже, у багатодоменному середовищі розподільник навантаження слід розглядати не лише як компонент інфраструктурного розгортання. У запропонованій інформаційній технології він виконує роль одного з ключових вузлів прийняття рішень. Через нього узгоджуються вимоги до продуктивності, кібербезпеки та керування ресурсами, а також забезпечується їх реалізація в робочому режимі. Таким чином, розподільник навантаження формує центральну ланку інтегрованого контуру, у межах якого вимірювання експлуатаційних параметрів, застосування захисних механізмів і керувальні дії щодо ємності кластера здійснюються узгоджено та з огляду на цільові показники якості обслуговування.

1.5. Загрози доступності та захист на рівні L7 у системах масового хостингу

Для систем масового вебхостингу, що одночасно підтримують десятки тисяч доменів, загрози доступності набувають системного масштабу , особливо коли такі системи обслуговують вебресурси з підвищеними вимогами до безперервності, зокрема пов'язані з критичною інфраструктурою. [1,2]. За умов спільного використання

обчислювальних ресурсів вплив інциденту не обмежується одним вебсайтом [10,65]. Навіть цілеспрямована дія, скерована на окремий ресурс, може спричинити дефіцит спільних компонентів інфраструктури [28,31,65]. До них належать точка входу, механізми встановлення захищених з'єднань, оперативна пам'ять як обмежувальний ресурс конкурентності, а також обчислювальні вузли, що виконують оброблення запитів [21,25]. У такій архітектурі доступність визначається не лише станом прикладного коду конкретного вебзастосунку. Вона залежить від стійкості спільного інфраструктурного контуру до пікових і аномальних режимів роботи та проявів шкідливої активності [63,65]. Тому забезпечення доступності виходить за межі локальних механізмів захисту окремого сайту та перетворюється на інфраструктурну функцію надавача послуг, яка має реалізовуватися на рівні кластера і точки входу, зважаючи на взаємний вплив орендарів [10,34].

Найбільшу небезпеку для описаного багатодоменного середовища становлять розподілені атаки на доступність, реалізовані на прикладному рівні, зокрема шторми HTTP-запитів [28,31]. Їхня критичність зумовлена тим, що навантаження формується не за рахунок суто обсягу трафіку, а через масове ініціювання операцій, які потребують обчислювальних ресурсів і створюють конкурентність у критичних компонентах інфраструктури. Унаслідок цього атакувальна активність швидко переводить систему в режим дефіциту ресурсу навіть тоді, коли мережеві канали не перевантажені [25]. На відміну від атак, що реалізовуються на мережевому та транспортному рівнях і мають виражений об'ємний характер, прикладні атаки використовують запити, які коректні з погляду протоколу. Такі запити проходять стандартні етапи оброблення, включно з встановленням з'єднання та ініціалізацією прикладної логіки. Вони також можуть відтворювати типові шаблони звернень, характерні для звичайних користувачів. Тому просте фільтрування за ознаками низькорівневого трафіку недостатнє, а протидія потребує аналізу поведінкових характеристик і контексту звернень на рівні прикладного протоколу [35,34].

Ключова небезпека прикладних атак зумовлена асиметрією обчислювальних витрат між стороною ініціатора та стороною, що обслуговує запит [28,33]. Для зловмисника вартість формування запиту мінімальна. Натомість інфраструктура провайдера змушена виконувати послідовність операцій, кожна з яких споживає обмежені ресурси [28,65]. До таких операцій належать встановлення захищеного з'єднання, виділення оперативної пам'яті для робочих процесів або контексту виконання [24,25], ініціалізація інтерпретатора прикладної платформи, звернення до підсистем зберігання даних і формування динамічної відповіді [15]. За високої частоти таких запитів система переходить у режим зростання конкурентності, а дефіцит оперативної пам'яті стає безпосередньою причиною деградації якості обслуговування та відмов [25,14]. Модулювання витрат пам'яті з боку атакуючої сторони та сервера що приймає запити наведено на рис. 1.12.

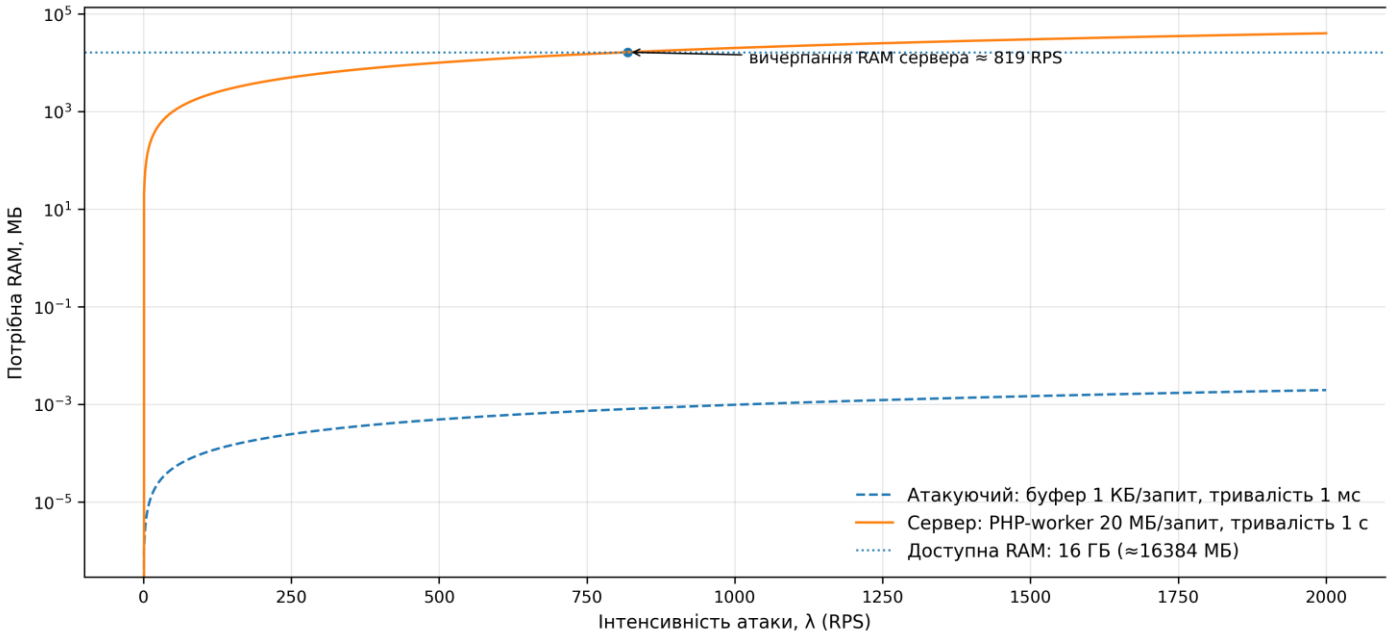

Рисунок 1.12 - Зростання витрат пам'яті унаслідок збільшення конкурентності запитів під час прикладної (L7) атаки: порівняння атакуючої сторони та сервера.

Окремий клас загроз становлять атаки, пов'язані з автоматизованим підбором облікових даних. У таких сценаріях навантаження формується не лише кількістю запитів, а й їхньою структурою. Повторювані звернення до механізмів автентифікації або до ресурсоємних кінцевих точок підвищують частку операцій, що потребують додаткової

перевірки, доступу до сховищ і створення стану сесії [78,32]. Це швидко збільшує кількість паралельних оброблень і підводить систему до межі за оперативною пам’яттю, що є критичним у масовому хостингу [24,25].

Особливість багатодоменного середовища полягає в тому, що такі загрози можуть не мати виражених ознак перевантаження на рівні окремого домену [10]. Атакувальна активність розподіляється між багатьма сайтами, тому для кожного з них інтенсивність звернень може залишатися відносно низькою та не викликати спрацювання локальних порогових механізмів [29]. Проте в сукупності ці дії формують критичне навантаження на спільні компоненти кластера [10,65]. Унаслідок цього зростає загальна конкурентність оброблення, підвищується споживання оперативної пам’яті та зменшується запас ресурсів, необхідний для підтримання заданих показників доступності й часу відповіді для легітимних користувачів [16,25].

Практичні механізми протидії загрозам на прикладному рівні зазвичай реалізовуються у вигляді багаторівневої системи, де кожен рівень вирішує окремий клас завдань і зменшує навантаження на наступні етапи оброблення. Така організація дає можливість поєднати прості швидкі перевірки з більш складними методами, які застосовуються лише до підозрілих або нетипових запитів [34]. На початковому рівні застосовуються механізми обмеження інтенсивності звернень. Вони встановлюють допустиму частоту запитів для мережевих адрес, груп адрес або окремих маршрутів і тим самим знижують вплив масових повторюваних звернень. Далі використовуються засоби фільтрації на основі правил, які аналізують запити за формальними ознаками та блокують типові шаблони зловмисної активності. До таких засобів належать міжмережеві екрани прикладного рівня, що порівнюють параметри запитів із сигнатурами відомих уразливостей і типових спроб експлуатації [79,80].

Наступний рівень орієнтований на підтвердження легітимності клієнта в ситуаціях, коли автоматизована активність маскується під звичайну поведінку. Для цього застосовуються механізми інтерактивної перевірки, побудовані за принципом «виклик-відповідь» (Captcha), які ускладнюють роботу автономних агентів [81]. Паралельно

можуть використовуватися репутаційні джерела, що містять відомості про мережеві адреси та інші ідентифікатори, пов’язані з попередніми інцидентами або підозрілими діями. Такі дані дають змогу швидко відсікати частину небажаного трафіку без детального аналізу кожного запиту [82].

Найвищий рівень захисту пов’язаний з аналізом поведінкових характеристик і виявленням аномалій, які не описуються простими правилами. Тут використовуються моделі, що оцінюють послідовність дій клієнта, зміну параметрів сесії, частоту повторень, розподіл запитів у часі та інші ознаки, характерні для автоматизованої активності. Для підвищення здатності виявляти складні та неочевидні сценарії застосовуються методи машинного навчання, які дають можливість узагальнювати поведінкові шаблони та виявляти відхилення від типових режимів роботи системи [35,83,84].

Водночас інтеграція прикладного захисту в масовий хостинг супроводжується принциповим компромісом між точністю виявлення загроз і продуктивністю інфраструктури [35,65]. Методи, що забезпечують високу якість фільтрації, як правило, потребують поглибленого аналізу змісту запиту та контексту взаємодії. До таких методів належать перевірки на основі складних шаблонів, зокрема з використанням регулярних виразів, а також класифікація запитів за допомогою моделей машинного навчання [80,83,84]. Обчислювальні витрати таких перевірок відчутні, особливо у середовищі з великою кількістю доменів і високою частотою звернень [35].

Якщо зазначений аналіз виконувати для кожного запиту в синхронному режимі, тобто залучати його безпосередньо до критичного шляху оброблення, підсистема безпеки починає визначати загальну пропускну здатність точки входу. За зростання інтенсивності трафіку накопичуються черги, збільшується час очікування та погіршуються показники, що контролюються угодами про рівень надання послуг. У такій ситуації механізм захисту фактично стає «вузьким місцем», незалежно від того, наскільки ефективно працюють обчислювальні вузли, що виконують прикладну логіку [35,16,14].

У масових системах це призводить до явища, коли сама процедура інспекції трафіку створює умови для відмови в обслуговуванні [35,14]. Затримки, пов’язані з перевірками, можуть перевищувати час, власне, оброблення запиту на бекенді, а сумарна конкурентність обмежується ресурсами точки входу. Такий режим доцільно трактувати як відмову в обслуговуванні, спричинену внутрішніми механізмами, оскільки деградація виникає не через обсяг зовнішнього трафіку як такий, а через надмірні витрати на синхронну перевірку кожного звернення [65,14]. Саме тому під час проєктування захисту для масового хостингу необхідно відокремлювати ресурсоємний аналіз від критичного шляху та забезпечувати його виконання у спосіб, що не погіршує якість обслуговування легітимних користувачів [35].

З огляду на зазначене, доцільним є архітектурне розмежування процесів фільтрації на два узгоджені тракти з різними вимогами до часу виконання [35]. Синхронний тракт має бути оптимізований на мінімальну затримку. У ньому рішення приймаються за простими правилами, які не потребують складних обчислень під час оброблення запиту. Такі правила мають бути попередньо підготовлені та зберігатися у структурі, що забезпечує швидкий доступ. Як наслідок, точка входу зберігає стабільну пропускну здатність і не формує додаткових черг для легітимного трафіку [16,14]. Натомість поглиблений аналіз, який бере до уваги поведінкові ознаки, статистичні закономірності та контекст взаємодії клієнта, доцільно винести в окремий асинхронний контур [35,83,84]. У цьому контурі допускаються більші витрати часу на обчислення, оскільки він не входить до критичного шляху обслуговування запиту [14]. Асинхронне оброблення може виконувати накопичення даних, побудову профілів, класифікацію запитів, виявлення ознак прикладних кіберзагроз та формування керувальних рішень у вигляді оновлень правил доступу для синхронного тракту. Така організація дає змогу використовувати складні методи аналізу без прямого впливу на затримки для легітимних користувачів [35]. Структурну схему інформаційної технології показано на рис. 1.13.

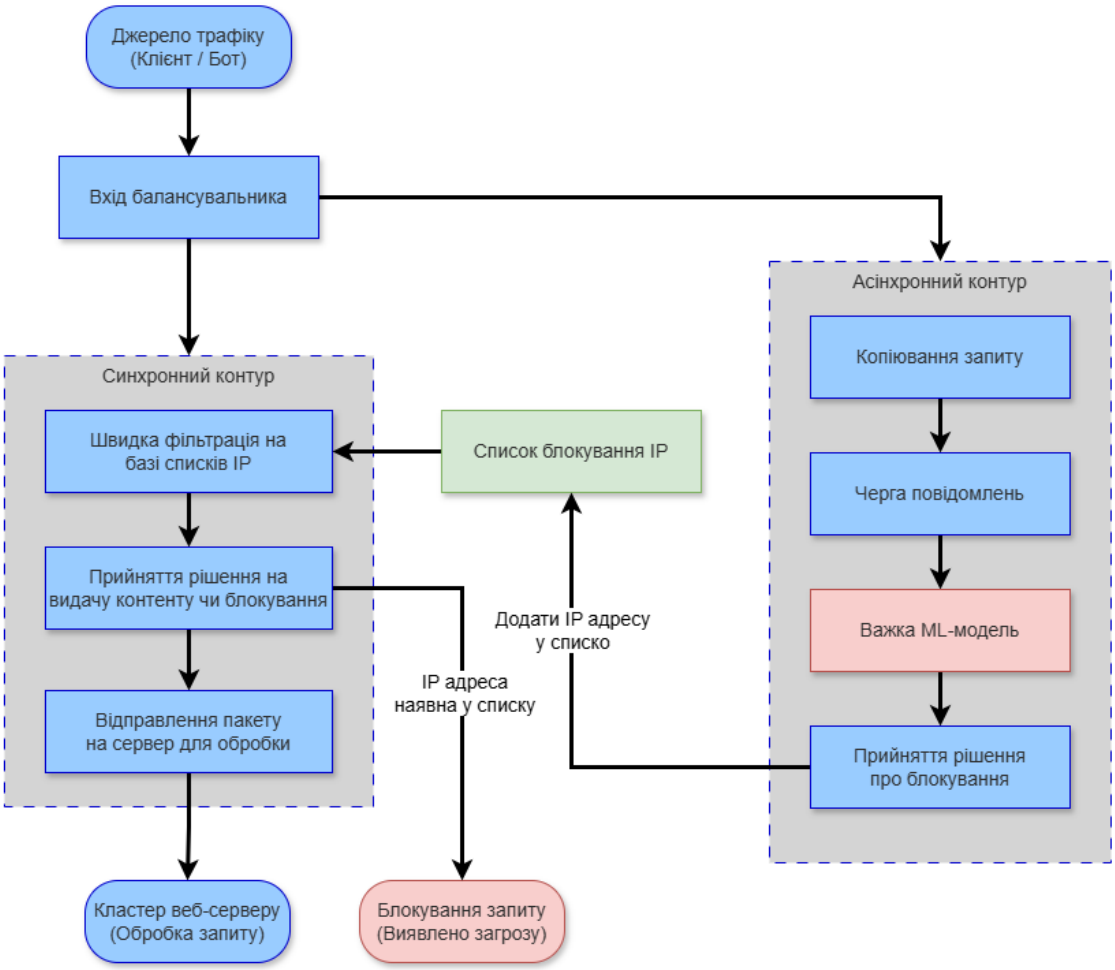

Рисунок 1.13 - Структурна схема інформаційної технології захисту ресурсів хостингу з розділенням синхронної фільтрації та асинхронного інтелектуального аналізу.

З позицій якості обслуговування цей підхід забезпечує прогнозовану швидкодію точки входу, оскільки затримка синхронного тракту визначається обмеженим набором простих операцій. Крім того, він підвищує стійкість системи до внутрішніх перевантажень аналітичних модулів. Навіть якщо асинхронний контур тимчасово не встигає обробляти потік подій або потребує масштабування, основний тракт продовжує працювати. У такому режимі система не блокує легітимний трафік через затримки аналізу [35,14]. Це відповідає принципу пропускання трафіку за замовчуванням у разі деградації аналітичної підсистеми, коли пріоритетом є збереження доступності сервісу та виконання часових вимог - принцип **Fail-Open** [77].

У публікації автора, присвяченій стійкості кластерного хостингу до розподілених атак, показано доцільність поєднання швидкодійних обмежувальних механізмів із відкладеним аналізом поведінкових характеристик джерел. У межах такого підходу первинне відсікання підозрілої активності виконується на вході системи за простими правилами, тоді як поглиблений аналіз формується окремо та використовується для уточнення керувальних рішень. Це дозволяє зменшити ризик деградації продуктивності через ресурсоємну інспекцію трафіку і водночас підвищити точність виявлення шкідливої активності завдяки поведінковим ознакам [30,83,84].

Зазначений прикладний механізм реалізовано також у запатентованому рішенні автора. У цій реалізації синхронний тракт забезпечує фільтрацію без додаткових затримок, використовуючи динамічні правила доступу, що застосовуються у критичному шляху обслуговування запиту. Асинхронний контур, своєю чергою, виконує аналіз і формує оновлення цих правил у робочому режимі. Важливо, що оновлення здійснюється без впливу на основний потік оброблення, тобто без блокування запитів і без погіршення часових характеристик для легітимних користувачів. Така організація забезпечує сумісність вимог безпеки з вимогами до доступності та передбачуваної швидкодії в масовому багатодоменному середовищі [35].

Отже, система захисту для багатодоменного середовища має спиратися на асинхронне оброблення телеметричних даних і відокремлення ресурсоємного аналізу від критичного шляху обслуговування запиту. Така організація дає можливість поєднати вимоги до стійкості проти прикладних атак із вимогами до передбачуваної швидкодії та доступності, не перетворюючи підсистему безпеки на «вузьке місце» інфраструктури [35,30,16,14]. Наведені положення визначають нефункціональні вимоги до запропонованої інформаційної технології та обґрунтовують вибір відповідних алгоритмічних рішень. Їх математичне обґрунтування подано в розділі 2, архітектурну реалізацію наведено в розділі 3, а експериментальну перевірку працездатності та відповідності заданим показникам якості — у розділі 4.

1.6. Висновки з огляду та постановка задачі дослідження

Проведений аналіз архітектурних підходів і методів керування обчислювальними ресурсами показує, що для масового хостингу слабо навантажених вебсайтів визначальним є не локальне підвищення ефективності окремого вузла або окремого сервісу. Критичною стає керована консолідація великої кількості стохастичних навантажень у межах спільного хмарного кластера. За таких умов об’єктом оптимізації виступає не одиничний елемент інфраструктури, а сукупний ресурсний пул, у якому потрібно забезпечити узгоджене використання пам’яті, обчислювальної потужності та пропускної здатності зважаючи на нерівномірність й непередбачуваність трафіку [8,23,63]. Водночас масштаб багатодоменного середовища виявляє принципову суперечність між двома групами вимог. З одного боку, економічна доцільність спонукає провайдера підвищувати щільність консолідації та зменшувати частку резерву, що простоює. З іншого боку, технічні умови експлуатації вимагають забезпечувати стабільну якість обслуговування, яка формалізується через показники SLA, а також підтримувати високу доступність сервісу в пікових і аномальних режимах. Отже, у завданні масового хостингу необхідно досягти такого режиму консолідації, за якого економічний ефект не супроводжується неконтрольованим зростанням ризиків деградації продуктивності та порушенням гарантованих параметрів якості [23,60,16,65].

Зменшення резерву обчислювальних потужностей безпосередньо підвищує економічну результативність інфраструктури. Водночас воно різко посилює ризики погіршення якості обслуговування. За дефіциту ресурсу зростає ймовірність збільшення часу відповіді, появи відмов в обслуговуванні та підвищення чутливості до прикладних кібератак. Це пов’язано з тим, що навіть короткочасні сплески конкурентності швидко вичерпують доступні ресурси, а наслідки такого вичерпання проявляються не лінійно, а у вигляді швидкого накопичення черг і каскадного погіршення часових характеристик [65,14]. Додаткові чинники формуються сучасними умовами експлуатації. По-перше, у масовому вебсередовищі домінує трафік, що передається із застосуванням шифрування, а отже на точку входу покладаються обчислювально затратні процедури встановлення

захищених з’єднань і вибору сертифікатів [19,21]. По-друге, багатодоменна інфраструктура потребує частих змін конфігурації, зокрема оновлення сертифікатів і правил доступу, причому такі зміни мають застосовуватися без зупинки сервісу та без переривання активних сесій [49,43,72]. Сукупно це означає, що навантаження на інфраструктуру формується не лише запитами користувачів, а й керувальними подіями, які супроводжують роботу середовища [58].

Унаслідок цього підвищуються вимоги до точки входу, тобто до розподільника навантаження, який виконує роль центрального вузла системи. Він має забезпечувати високу пропускну здатність і стабільні затримки в умовах шифрованого трафіку. Одночасно він повинен підтримувати безперервну керованість, тобто можливість оперативного застосування змін конфігурації в робочому режимі [49,21]. Не менш важливою є спостережуваність, оскільки саме точка входу формує узгоджений набір метрик, необхідних для контролю якості обслуговування і для роботи контурів масштабування та захисту [23,16]. Отже, розподільник навантаження в масовому хостингу є не лише компонентом маршрутизації, а ключовим елементом, від характеристик якого залежить здатність інфраструктури одночасно досягати економічної ефективності та виконувати задані показники якості і доступності [23,16].

Наведений аналіз дає підстави сформулювати ключову проблему дослідження. У хмарному кластері, призначеному для динамічного хостингу, існує об’єктивна потреба підвищити ефективність використання обчислювальних ресурсів і зменшити частку непродуктивного резервування. Водночас досягнення такого підвищення не може супроводжуватися погіршенням якості обслуговування, що формалізується показниками SLA. Так само воно не повинно знижувати стійкість інфраструктури до загроз доступності, які реалізовуються на прикладному рівні. Отже, завдання полягає у пошуку таких архітектурних і алгоритмічних рішень, які забезпечують ресурсну ефективність кластера за збереження заданих гарантій якості та здатності протидіяти прикладним атакам [8,23,16,31].

У межах дисертаційної роботи як критично обмежувальний ресурс розглядається передусім оперативна пам’ять. Саме її доступний обсяг жорстко визначає граничну кількість одночасно активних процесів і контекстів, що забезпечують оброблення запитів у середовищі масового хостингу. За цих умов ресурсна ефективність інфраструктури безпосередньо залежить від того, наскільки точно узгоджено рівень конкурентності з фактичними можливостями пам’яті та з вимогами до якості обслуговування [25,85]. Відповідно, оптимізаційна постановка завдання полягає у визначенні керованої ємності кластера в часі та у розробленні алгоритмів керування, які забезпечують її адаптацію до стохастичних коливань навантаження. Такі алгоритми мають зменшувати частку непродуктивного резервування ресурсу, насамперед пам’яті, і водночас забезпечувати формалізовані гарантії дотримання заданого рівня якості обслуговування. У цьому контексті вимоги до якості мають бути подані у математичній формі, що дає змогу пов’язати параметри трафіку, характеристики оброблення та доступну ємність системи з імовірнісними обмеженнями, які відповідають показникам SLA [16,14].

На підставі проведеного аналізу в дисертаційній роботі поставлено науково-прикладне завдання — розробити інформаційну технологію оптимізації використання обчислювальних ресурсів у хмарних кластерах для динамічного хостингу веб-сервісів, яка забезпечує дотримання вимог угод про рівень надання послуг в умовах стохастичного багатодоменного навантаження та створює умови для застосування прикладних механізмів захисту без погіршення швидкодії основного тракту оброблення легітимних запитів.

Для виконання поставленого завдання в роботі визначено такі основні напрями дослідження:

1. Розроблення математичної моделі планування ємності хмарного кластера для динамічного вебхостингу.
2. Розроблення методу асинхронного контуру прикладного захисту.

3. Удосконалення методу визначення необхідної конфігурації кластера та керування його масштабуванням.
4. Розроблення методу оцінювання ресурсної ефективності консолідації.
5. Обґрунтування архітектурних принципів реалізації запропонованої інформаційної технології та підготовлення дослідницького програмного прототипу для її експериментальної перевірки.

Виконання сформульованих завдань визначає загальну логіку та структуру дисертаційної роботи, яку узагальнено у вигляді структурно-логічної схеми на рис. 1.14.

Відповідно до наведених етапів дослідження, у розділі 2 подано розроблені математичні моделі й алгоритми, що описують планування ємності та адаптивне керування ресурсами кластера з огляду на стохастичний характер навантаження. Також у цьому розділі подано метод функціонування асинхронного контуру аналізу трафіку, який використовується для формування керувальних рішень без впливу на критичний шлях оброблення запитів.

У розділі 3 подано системний опис запропонованої інформаційної технології та архітектури її реалізації. Розподільник навантаження розглянуто як центральний елемент точки входу, що забезпечує термінування захищених з’єднань, вибір домену та застосування керувальних впливів щодо розподілу ресурсу. Механізми захисту реалізовано як поєднання швидкого синхронного фільтрування за динамічними правилами та асинхронного інтелектуального аналізу, результати якого використовуються для оновлення цих правил у робочому режимі.

У розділі 4 наведено результати програмної реалізації дослідницького прототипу та експериментальної перевірки запропонованих моделей і архітектурних рішень у динамічних сценаріях, характерних для масового багатодоменного хостингу. Завдяки одержаним результатам стало можливим оцінити практичну придатність розробленої технології та узгодженість теоретичних положень із експериментальними спостереженнями.

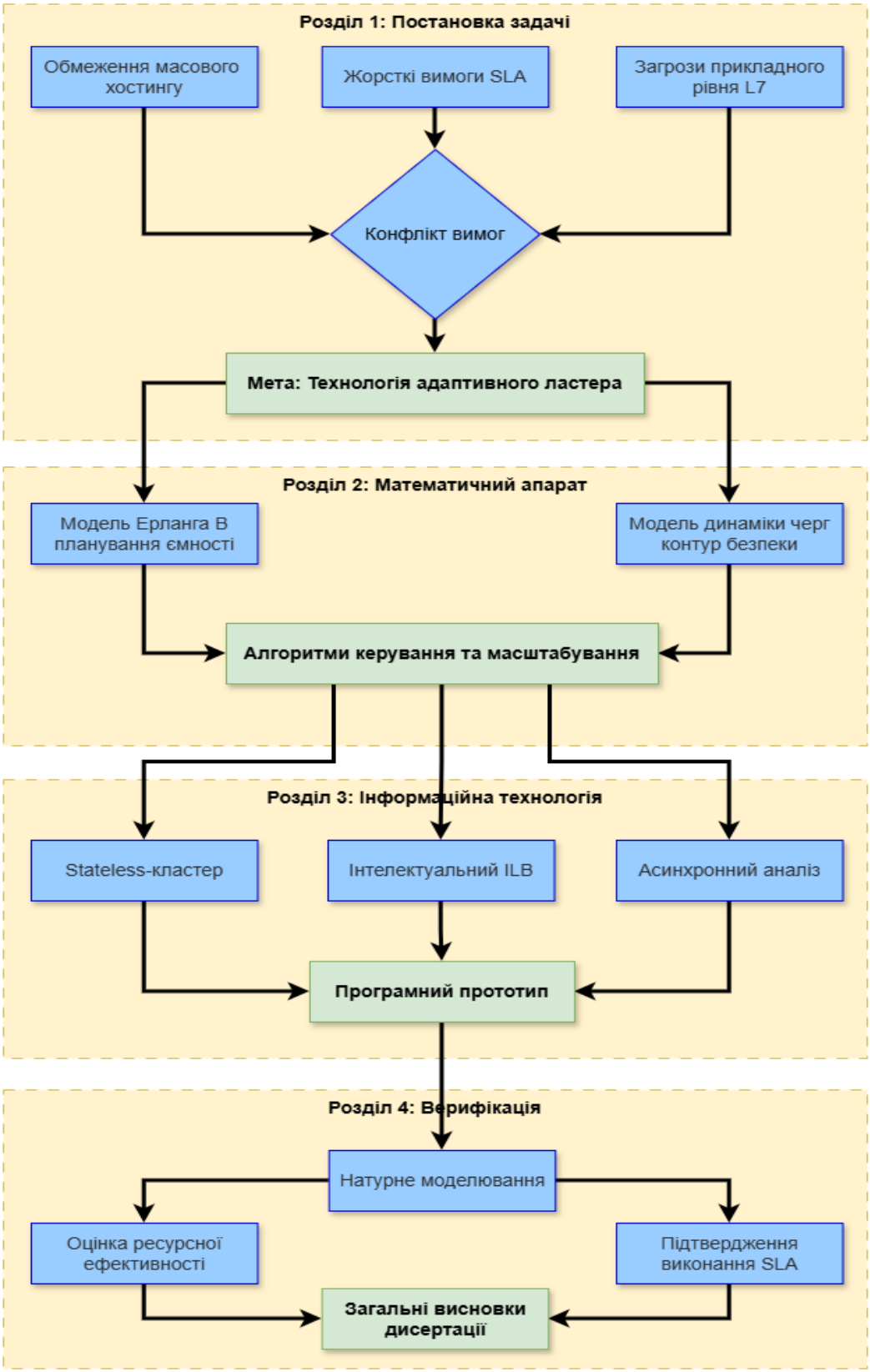

Рисунок 1.14 - Структурно-логічна схема розроблення інформаційної технології адаптивного хостингу.

1.7. Висновки до першого розділу

У першому розділі дисертації проаналізовано стан проблеми оптимізації використання обчислювальних ресурсів у хмарних кластерах для динамічного хостингу веб-сервісів та обґрунтовано актуальність дослідження для сегмента масового розміщення слабо навантажених сайтів. Показано, що для такого середовища визначальними є не пікові характеристики окремого вебресурсу, а ефективність спільного використання ресурсного пулу за умов стохастичного, нерівномірного та багатодоменного навантаження. Встановлено, що економічна доцільність масового хостингу безпосередньо залежить від рівня консолідації ресурсів, однак збільшення щільності розміщення супроводжується зростанням ризиків деградації продуктивності та порушення показників SLA.

У ході аналізу наявних архітектурних підходів встановлено, що класичний спільний хостинг, віртуальні машини, контейнеризація та безсерверні моделі не забезпечують одночасного досягнення трьох критично важливих вимог: високої щільності консолідації, передбачуваної якості обслуговування та прийнятних накладних витрат у масштабі десятків тисяч доменів. На цій підставі обґрунтовано доцільність спеціалізованої кластерної архітектури динамічного хостингу, у якій керування ресурсами, балансування навантаження та захист прикладного рівня інтегруються в єдиний керований контур.

Показано, що центральним елементом такої архітектури є балансувальник навантаження прикладного рівня, який виконує не лише функції маршрутизації, а й термінування TLS-з’єднань, вибору домену, збору телеметрії та ініціювання керувальних впливів. Доведено, що саме точка входу значною мірою визначає масштабованість, спостережуваність і стійкість усієї системи, оскільки на ній концентруються як обчислювально затратні криптографічні операції, так і динамічні зміни конфігурації, характерні для багатодоменного середовища.

Окремо встановлено, що суттєвою загрозою для масового хостингу є атаки прикладного рівня, передусім HTTP flood і автоматизовані brute-force сценарії, які

створюють асиметричне ресурсне навантаження та можуть призводити до дефіциту оперативної пам’яті й деградації сервісу навіть без перевантаження мережевих каналів. Унаслідок цього обґрунтовано, що традиційне синхронне виконання ресурсоємного аналізу трафіку в критичному тракті оброблення запитів є неприйнятним, оскільки саме може перетворитися на вузьке місце системи. Це зумовлює потребу в асинхронному контурі прикладного захисту, який забезпечує поглиблений аналіз та формування керувальних правил без погіршення швидкодії легітимного трафіку.

За результатами огляду зроблено висновок, що ключовим обмежувальним ресурсом у задачі масового динамічного хостингу доцільно розглядати оперативну пам’ять, оскільки саме вона визначає граничну кількість одночасних обслуговувань і рівень допустимої конкурентності. Відповідно, задача оптимізації має формулюватися як задача визначення мінімально достатньої керованої ємності кластера та правил її адаптації до стохастичних коливань навантаження за умови дотримання заданих вимог SLA. Це створює підстави для подальшого використання математичних моделей теорії масового обслуговування та втратних систем для планування ємності й керування масштабуванням.

Таким чином, у першому розділі сформовано теоретико-прикладне підґрунтя дисертаційного дослідження та визначено основне науково-прикладне завдання: розроблення інформаційної технології оптимізації використання обчислювальних ресурсів у хмарному кластері для динамічного хостингу веб-сервісів, яка поєднує адаптивне управління ресурсами, ресурсну ефективність, дотримання показників якості обслуговування та кібербезпекову стійкість до загроз доступності на прикладному рівні. Отримані висновки безпосередньо визначають зміст наступних розділів роботи, присвячених математичному моделюванню, архітектурній реалізації та експериментальній перевірці запропонованих рішень.

РОЗДІЛ 2. МАТЕМАТИЧНІ МОДЕЛІ ПЛАНУВАННЯ ЄМНОСТІ ТА АДАПТИВНОГО УПРАВЛІННЯ РЕСУРСАМИ ХМАРНОГО КЛАСТЕРА

У попередньому розділі обґрунтовано, що для досягнення високої щільності консолідації вебресурсів без погіршення якості обслуговування доцільно перейти до архітектури динамічного хмарного кластера. В умовах масового хостингу критично обмежувальним ресурсом виступає оперативна пам’ять, оскільки її обсяг безпосередньо визначає граничну кількість одночасно активних запитів. За таких умов підходи до масштабування, що ґрунтуються на усереднених показниках завантаження процесора, не забезпечують своєчасного виявлення перевантажень і не відображають реальної причини деградації якості. Отже, виникає потреба у строгому математичному апараті, який дає змогу кількісно пов’язати стохастичний характер вхідного вебтрафіку з апаратними обмеженнями системи та з імовірнісними вимогами угод про рівень надання послуг. Такий апарат має забезпечувати аналітичне визначення необхідної ємності кластера і формалізувати умови, за яких задані показники якості обслуговування зберігаються в умовах нерівномірного навантаження та короткочасних пікових відхилень.

Мета цього розділу: формалізація завдання керування обчислювальними ресурсами та побудова комплексу математичних моделей, які становлять теоретичну основу розроблюваної інформаційної технології. Для досягнення цієї мети інфраструктуру динамічного хостингу подано у вигляді багатоканальної системи масового обслуговування з відмовами, у якій кількість каналів визначається доступною конкурентністю оброблення запитів [14,67,68].

Основою математичного апарату для планування необхідної ємності кластера обрано формулу Ерланга B. Вона дозволяє визначати мінімально достатню кількість серверних вузлів або еквівалентну кількість каналів обслуговування, за яких імовірність відмови в обслуговуванні не перевищує заданого граничного значення [14,67,68]. Така постановка дає змогу перейти від інтуїтивного резервування ресурсу до

формалізованого вибору базової ємності кластера, зважаючи на стохастичний характер вхідного потоку та вимог до якості обслуговування.

Окремої формалізації потребує асинхронний контур захисту від прикладних атак. У межах дослідження він подається як система черг із керованою продуктивністю, параметри якої змінюються залежно від поточного навантаження та доступного ресурсу. Зв допомогою такої моделі можна описати умови, за яких ресурсоємний аналіз трафіку виконується із затримкою, не входить до критичного шляху оброблення запитів і не створює блокування основного потоку легітимного трафіку [35,14]. Це забезпечує можливість поєднання вимог безпеки з вимогами до передбачуваної швидкодії точки входу.

Матеріал розділу подано за логікою переходу від формалізації цільової функції та системи обмежень (підрозділ 2.1) до алгоритмічного виконання завдання визначення необхідної ємності кластера із застосуванням рекурентних методів обчислення та механізму гістерезису (підрозділ 2.2). Далі в підрозділі 2.3 розроблено математичну модель динаміки черг для підсистеми кіберзахисту. Завершується розділ формуванням методу кількісного оцінювання економічної та ресурсної ефективності запропонованого підходу порівняно з класичною фрагментованою моделлю хостингу (підрозділ 2.4). Структурно-логічний взаємозв’язок комплексу математичних моделей показана на рис. 2.1.

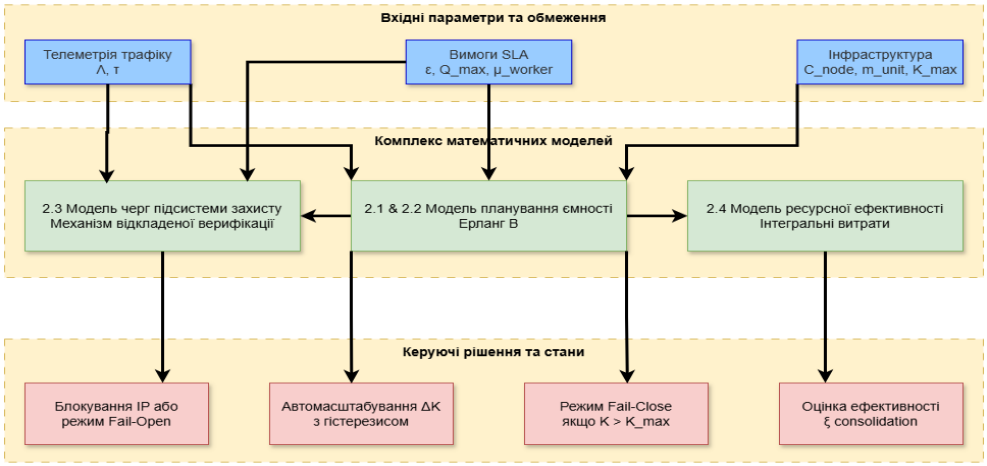

Рисунок 2.1 - Структурно-логічний взаємозв’язок комплексу математичних моделей керування обчислювальними ресурсами хмарного кластера.

2.1. Формалізація завдання адаптивного управління ресурсами у Stateless-кластері

Для побудови математичної моделі розглянемо систему хмарного хостингу, що забезпечує обслуговування множини вебресурсів:

$$W = \{w_1, w_2, \ldots, w_n\}$$

З огляду на вибір архітектурного підходу без збереження стану на обчислювальних вузлах, у межах моделі не розглядається класичне завдання жорсткого закріплення ресурсів, за якого кожному вебресурсу w_i відповідає конкретний фізичний сервер s_j. Натомість систему доцільно подати як єдиний обчислювальний пул, на вхід якого надходить агрегований стохастичний потік запитів:

$$\Lambda(t) = \sum_{i=1}^{n} \lambda_i\,(t)$$

де $\lambda_i(t)$ — інтенсивність надходження HTTP-запитів до i-го сайту в момент часу t.

2.1.1. Вибір базової одиниці ресурсу

Відповідно до результатів аналізу, наведених у розділі 1, обмежувальним ресурсом під час оброблення вебзапитів є оперативна пам’ять, що використовується процесами середовища виконання та інтерпретації прикладної логіки (наприклад, PHP-FPM або WSGI). У межах моделі вводиться поняття базового кванта пам’яті (m_{unit}), який відповідає обсягу оперативної пам’яті, необхідному для обслуговування однієї активної сесії, тобто одного конкурентного запиту [25,85]. Далі, спираючись на галузеві рекомендації та результати попередніх досліджень, візьмемо [24,25,85]:

$$m_{unit} \approx 256 \text{ MiB}$$

2.1.2. Параметри обчислювального вузла

Кластер розглядається як сукупність однотипних віртуальних серверів зі змінною кількістю активних вузлів. Ємність одного вузла C_{node} визначається максимальною

кількістю одночасних запитів, які він здатен обробляти без переходу до підкачки, тобто без використання swap-пам’яті:

$$C_{node} = \left\lfloor \frac{RAM_{total} - RAM_{os}}{m_{unit}} \right\rfloor$$

де

- RAM_{total} — загальна пам’ять сервера (MiB);
- RAM_{os} — резерв під операційну систему (MiB).

Приклад. Для сервера з 16 GiB RAM, $C_{node} \approx (16384 - 1024)/256 \approx 60$ одночасних запитів.

2.1.3. Цільова функція

Нехай $K(t)$ позначає кількість активних серверів кластера в момент часу t. Витрати експлуатації системи за період T визначаються вартістю оренди обчислювальних вузлів, що забезпечують оброблення запитів. Окрім змінної складової, враховуються постійні накладні витрати на інфраструктуру керування та балансування, які для цієї постановки вважаються сталими: ($Cost_{overhead} = const$). У хмарному середовищі вартість оренди береться лінійно залежною від обсягу використаних ресурсів, зокрема оперативної пам’яті. За таких припущень завдання оптимізації формулюємо як мінімізацію інтегрального показника споживання ресурсу оперативної пам’яті за період T:

$$F = \int_0^T (K(t) \cdot RAM_{total} + RAM_{overhead})\, dt \to \min$$

де

- RAM_{total} — повний обсяг пам’яті одного сервера;
- $RAM_{overhead}$ — повний обсяг пам’яті необхідний для інфраструктури керування та балансування.

$$RAM_{overhead} = RAM_{ILB} \cdot N_{ILB}$$

де

- RAM_{ILB} — повний обсяг пам’яті одного сервера балансувальника навантаження;
- N_{ILB} — кількість серверів балансувальника в системі.

У подальших розрахунках беремо $N_{ILB} = 2$, що відображає резервування точки входу. Це підвищує точність оцінювання сумарних витрат ресурсу, хоча не змінює логіку визначення $K(t)$, оскільки накладна складова стала.

2.1.4. Обмеження

Мінімізація функції F здійснюється за наявності системи обмежень, що відображає вимоги до якості обслуговування, стійкості підсистеми захисту, а також бюджетні й технічні обмеження інфраструктури.

1. Бюджетне обмеження.

Кількість активних вузлів кластера обмежується адміністративними межами. Верхня межа K_{max} запроваджується для запобігання неконтрольованому зростанню витрат, зокрема у випадках аномальних навантажень або атак, що можуть ініціювати надмірне масштабування. Нижня межа K_{min} визначається вимогами відмовостійкості й має бути не меншою за 2, щоб забезпечити працездатність системи у разі відмови окремого вузла:

$$K_{min} \leq K(t) \leq K_{max}$$

2. Обмеження за ймовірністю відмови (SLA).

За умови $K(t) < K_{max}$ кількість активних серверів $K(t)$ має бути достатньою для того, щоб імовірність відхилення запиту P_{block} через відсутність вільних пам’яттєвих слотів не перевищувала заданого граничного рівня ϵ. Як приклад можна взяти $\epsilon = 0{,}001$, що відповідає цільовій доступності 99,9%. У межах втратної моделі ця вимога формалізується за допомогою формули Ерланга B:

$$P_{block}(A(t), M(t)) = \frac{\frac{A^M}{M!}}{\sum_{i=0}^{M} \frac{A^i}{i!}} \leq \epsilon$$

де параметри моделі розраховуються в реальному часі:

- $M = K(t) \cdot C_{node}$ — загальна ємність кластера (сумарна кількість доступних слотів пам'яті на всіх активних серверах);
- $A(t) = \Lambda(t) \cdot \tau(t)$ — інтенсивність вхідного навантаження в Ерлангах, є безрозмірною та чисельно відповідає очікуваній кількості одночасно активних запитів (зайнятих слотів пам'яті) в системі;
- $\tau(t)$ — середній час утримання ресурсу (оброблення запиту), який є динамічною величиною і розраховується балансувальником у реальному часі методом експоненційного ковзного середнього на основі статистики завершених запитів за інтервал Δt, для згладжування короткочасних сплесків.

$$\tau(t) = \alpha \cdot \min(t_{last}, t_{cap}) + (1 - \alpha) \cdot \tau(t - 1)$$

де

- t_{last} — виміряний час виконання останнього завершеного запиту;
- t_{cap} — гранично допустимий час врахування запиту. Використання функції $\min()$ відсікає статистичні викиди та запобігає штучному «роздуванню показника $\tau(t)$ під час атак на кшталт "Slowloris" або HTTP-flood важкими запитами на окремі компоненти;
- $\tau(t - 1)$ — попереднє значення середнього часу;
- α — безрозмірний коефіцієнт згладжування ($0 < \alpha < 1$), що визначає чутливість системи до змін (емпірично рекомендовано $\alpha \approx 0.05$).

Примітка (Режим перевантаження). Якщо розрахункова потреба в ресурсі перевищує адміністративну межу, тобто $K_{req} > K_{max}$, система переходить у режим **Fail-Close**, який передбачає примусове обмеження вхідного потоку. У цьому режимі балансувальник відхиляє надлишкові запити, повертаючи відповідь HTTP 503 (Service

Unavailable) [77,86]. Пріоритет надається завершенню вже встановлених сесій і стабілізації роботи інфраструктури. Паралельно асинхронний контур продовжує аналізувати наявний потік телеметрії з використанням доступних обчислювальних потужностей для ідентифікації джерел аномальної активності та формування оновлень правил доступу. У роботі використовуються два різні режими деградації, що належать до різних контурів керування. Режим **Fail-Close** застосовується лише в контурі забезпечення SLA у разі, коли розрахункова потреба перевищує адміністративну межу ($K_{req} > K_{max}$), і означає примусове обмеження вхідного потоку для захисту працездатності інфраструктури. Натомість режим **Fail-Open** описаний у п. 3.4.4, стосується виключно асинхронного аналітичного контуру. У разі переповнення черги або недоступності ML Worker нові події можуть не аналізуватися, але основний тракт обслуговування продовжує роботу без блокування легітимного трафіку.

3. Обмеження стійкості асинхронної системи захисту.

Підсистема захисту моделюється як система масового обслуговування з буфером обмеженої ємності. Для забезпечення стабільної роботи та недопущення переповнення пам’яті балансувальника мають виконуватися дві умови.

Умова балансу потоків. Інтенсивність потоку запитів, що надходять на поглиблений аналіз, не повинна перевищувати пропускну здатність модуля захисту:

$$\Lambda(t) \cdot (1 - \eta(t)) \leq \mu_{worker} \cdot N_{workers}$$

де

- $\Lambda(t)$ — інтенсивність вхідного навантаження, зап./с;
- $\eta(t) \in [0,1]$ — коефіцієнт ефективності попередньої фільтрації. Він показує частку трафіку, яка класифікується миттєво за допомогою списків контролю доступу (ACL) у пам’яті балансувальника і не потребує звернення до нейромережі;
- μ_{worker} — продуктивність одного ML-воркера, зап./с;
- $N_{workers}$ — кількість паралельних процесів аналізу.

Умова обмеження черги. Довжина черги відкладеного аналізу $Q(t)$ не повинна перевищувати максимально допустиме значення Q_{max}:

$$Q(t) \leq Q_{max}$$

Примітка. У разі порушення умови балансу потоків, довжина черги $Q(t)$ починає зростати. Після досягнення граничного значення $Q(t) = Q_{max}$ система переходить у режим **Fail-Open**, у якому надлишкові запити пропускаються без поглибленого аналізу. Такий режим застосовується для збереження доступності сервісу та недопущення блокування основного тракту оброблення легітимного трафіку [65,77].

2.2. Математична модель розрахунку необхідної ємності кластера

У попередньому підрозділі умову якості обслуговування було задано у вигляді обмеження на ймовірність відмови, сформульованого через формулу Ерланга B. Для практичної реалізації модуля автоматичного масштабування на основі стохастичної моделі навантаження необхідно розв'язати обернену задачу. Потрібно визначити мінімально необхідну кількість серверів $K_{req}(t)$, яка під час поточного навантаження $A(t)$ забезпечує ймовірність відмови $P_{block} \leq \epsilon$.

Оскільки залежність Ерланга B нелінійна відносно K і не допускає отримання розв'язку в замкненому вигляді, обчислення K_{req} доцільно виконувати чисельно. Для застосування в реальному часі пропонується використати ітераційний метод, який послідовно підбирає найменше значення K_i, що задовольняє задане обмеження на $P_{block} \leq \epsilon$.

2.2.1. Рекурентний метод розрахунку ймовірностей

Пряме обчислення за формулою, наведеною в п. 2.1.4, є обчислювально складним і має практичні обмеження для реалізації в контурі керування. По-перше, використання факторіалів ($K!$) та степеневих виразів у разі зростання K призводить до числової нестійкості обчислень. Зокрема, виникають переповнення, втрата значущих розрядів і накопичення похибок під час операцій із числами з плаваючою комою. По-друге, спроба обмежувати допустимі значення K на рівні алгоритму суперечить вимозі

універсальності розроблюваної технології. Загалом вона має бути придатною не лише для кластерів із невеликою кількістю вузлів, а й для дрібніших обчислювальних одиниць, наприклад мікроконтейнерів, де кількість активних екземплярів може досягати сотень.

Для забезпечення чисельної стійкості та підвищення продуктивності обчислень доцільно застосувати рекурентне співвідношення, яке дає змогу визначати ймовірність втрат $E_M(A)$ для M каналів обслуговування на основі відомого значення для $M-1$ каналів:

$$E_M(A) = \frac{A \cdot E_{M-1}(A)}{M + A \cdot E_{M-1}(A)}$$

де:

- $E_M(A)$ — імовірність відмови при наявності M каналів обслуговування (еквівалент P_{block});
- Початкова умова: $E_0(A) = 1$ (якщо каналів обслуговування 0, імовірність втрати запиту = 100%).

Такий підхід гарантує, що всі проміжні значення залишаються в межах інтервалу [0,1]. Це усуває ризик переповнення та істотно зменшує втрату точності незалежно від масштабу кластера. Окрім того, рекурентне обчислення не потребує використання факторіалів і чисельно стійке, що робить його придатним для високочастотного перерахунку необхідної ємності в контурі керування.

2.2.2. Алгоритм пошуку оптимальної конфігурації

Завдання планування ємності формулюється як пошук найменшого цілого значення кількості слотів M, для якого виконується умова $E_M(A) \leq \epsilon$. З огляду на лінійну обчислювальну складність рекурентного обчислення, визначення необхідної місткості доцільно виконувати послідовною ітерацією, починаючи з базового значення M і збільшуючи його доти, доки не буде виконано обмеження за SLA. Такий порядок пошуку забезпечує мінімальність знайденого значення M для кожного поточного рівня запропонованого навантаження. Отже, за будь-якої зміни A алгоритм повертає

найменшу місткість, достатню для дотримання заданого порога ϵ, що відповідає мінімально необхідному обсягу ресурсу за сформульованим критерієм.

1. **Вхідні дані:**

   $A_{current}$ — поточне навантаження (Ерланг);

   ϵ — цільовий SLA, перетворений у ймовірність відмови (наприклад 99,9% → 0.001).

2. **Ітераційний пошук**. У циклі, починаючи з $M = 0$, розраховується ймовірність відмови $E_M(A_{current})$ за рекурентною формулою. Щойно досягається цільова умова $E_M(A) \leq \epsilon$, цикл припиняється, а поточне значення M фіксується як мінімально необхідна кількість слотів M_{req}.

3. **Агрегування ємності**. Перехід від кількості слотів до кількості фізичних серверів виконується шляхом ділення M_{req} на ємність одного вузла з округленням у більший бік:

$$K_{req}(t) = \left\lceil \frac{M_{req}(t)}{C_{node}} \right\rceil$$

4. **Керуючий вплив**. Рішення про масштабування приймається на основі різниці між поточною кількістю активних серверів K_{curr} та розрахунковою необхідною кількістю K_{req}:

$$\Delta K = K_{req} - K_{curr}$$

- Якщо $\Delta K > 0$, до кластера додається ΔK серверів (з урахуванням верхнього обмеження K_{max}, описане в п. 2.1.4).
- Якщо $\Delta K < 0$, з кластера виводиться $|\Delta K|$ серверів (з урахуванням нижнього обмеження K_{min} (п. 2.1.4)) та умови гістерезису, описані у наступному пункті.
- Якщо $\Delta K = 0$, розмір кластера залишається без змін.

Отже, для кожного поточного значення навантаження алгоритм повертає не довільну допустиму, а мінімально достатню конфігурацію кластера, яка ще задовольняє обмеження за SLA. Саме ця властивість забезпечує локальну мінімізацію поточного

ресурсу $K(t)$, а при розгляді процесу в часі — процедурну реалізацію мінімізації цільової функції F, введеної в п. 2.1.3.

2.2.3. Стабілізація контуру керування (Гістерезис)

Для забезпечення стійкості процесу регулювання та запобігання автоколиванням, тобто частим перемиканням між режимами збільшення і зменшення кластера за незначних коливань навантаження, у контур керування вводиться зона нечутливості, або гістерезис.

Умова для зменшення кількості серверів модифікується введенням коефіцієнта запасу δ (наприклад, $\delta = 0.1$, що відповідає запасу міцності у 10% відносно гранично допустимої ймовірності відмови ϵ). Зменшення ресурсу допускається лише тоді, коли прогнозована ймовірність відмови після відключення частини вузлів залишається в допустимих межах із заданим запасом:

$$E_{M-C_{node}}(A) \leq \epsilon \cdot (1 - \delta)$$

Це забезпечує, що система не вивільнятиме ресурси в ситуації, коли з високою ймовірністю їх доведеться повторно залучити вже в наступному циклі керування.

2.2.4. Моделювання інерційності контуру керування

Важливим аспектом адаптивного керування ресурсами у хмарному середовищі є врахування інерційності керуючого контуру. Процес горизонтального масштабування не є миттєвим: від моменту фіксації зростання навантаження до фактичного введення нових обчислювальних потужностей в експлуатацію проходить певний час.Загальний час реакції системи (запізнювання керуючого впливу) τ_{react} моделюється як сума трьох основних часових компонентів:

$$\tau_{react} \approx \tau_{smooth} + \tau_{recalc} + \tau_{boot}$$

де

- τ_{smooth} — період згладжування метрик. Це час, необхідний балансувальнику для накопичення статистичних даних та обчислення експоненційного ковзного середнього інтенсивності запитів. Для забезпечення 100% гарантії виконання SLA

99,9% використовується песимістична оцінка, що дорівнює повній ширині вікна згладжування (замість теоретичного середнього часу реакції).

- τ_{recalc} — час обчислення керуючого впливу. Відображає затримку на виконання ітеративного алгоритму розрахунку необхідної ємності за B-формулою Ерланга та передавання команди до API хмарного провайдера.
- τ_{boot} — час ініціалізації нового обчислювального вузла. Найтриваліший етап, який передбачає виділення ресурсів гіпервізором, завантаження гостьової ОС, запуск середовища виконання та реєстрацію вузла на балансувальнику.

Наявність інерційного запізнювання τ_{react} накладає жорсткі обмеження на здатність кластера поглинати раптові сплески трафіку. Для аналітичного оцінювання динамічної стійкості кластера далі вводяться дві допоміжні математичні величини, які використовуються насамперед для інтерпретації перехідних режимів і впливу інерційності контуру керування, а не як безпосередні керуючі параметри алгоритму масштабування.

Безпечна швидкість зростання навантаження (r_{safe} **)**. Ця величина задає консервативну аналітичну оцінку максимально допустимої похідної інтенсивності вхідного потоку в часі, за якої кластер встигне ввести в експлуатацію нові сервери до моменту вичерпання наявних вільних ресурсів. Якщо позначити через C_n граничне навантаження кластера з n активними серверами, а через λ_0 - поточне навантаження, то така оцінка набуває вигляду:

$$r_{safe}(n, \lambda_0) = \frac{C_n - \lambda_0}{\tau_{react}}$$

Якщо фактична швидкість зростання вхідного трафіку перевищує значення r_{safe}, потік вичерпає наявний резерв швидше, ніж завершиться період ініціалізації. У такому разі система неминуче перейде у стан перевантаження, що призведе до відкидання запитів із порушенням SLA.

Темп приросту місткості кластера (r_{cap}). Окремо вводиться допоміжна оцінка середнього темпу приросту загальної місткості кластера внаслідок додавання одного нового сервера:

$$r_{cap}(n) \approx \frac{C_{n+1} - C_n}{\tau_{react}}$$

Ця величина не є межею безвтратного зростання навантаження, а характеризує лише загальну швидкість, з якою кластер нарощує свою пропускну здатність після спрацювання умови додавання сервера. На практиці фактична поява втрат залежить також від поточного рівня навантаження, кількості вже активних вузлів, моменту спрацювання порога масштабування та розміру резерву, тому подальша перевірка цих ефектів у роботі виконується натурними експериментами у п. 4.5.4.

Зазначені математичні залежності доводять, що для систем із високою інерційністю ($\tau_{react} > 1$ хв) мінімізація втрат та жорстке збереження SLA при стрімких сплесках трафіку можливі лише двома способами:

1. Штучним збільшенням чисельника у формулі r_{safe} завдяки постійному утриманню надлишкового резерву. Великі значення резерву можуть суперечити критерію економічної ефективності.
2. Використанням архітектурних рішень для виявлення аномалій. Саме математична умова утримання SLA обґрунтовує абсолютну необхідність впровадження асинхронного контуру захисту з чергами відкладеного аналізу (Fail-Open), здатного виявити загрозу та відсікти нелегітимний надлишковий трафік протягом часу τ_{react}.

2.3. Модель черг для асинхронної системи захисту (L7)

У межах розробленої архітектури підсистему захисту винесено в окремий асинхронний контур. Це забезпечує розділення основного потоку обслуговування легітимних користувачів і потоку поглибленого аналізу потенційних загроз. У математичній постановці такий контур моделюється як багатоканальна система

масового обслуговування з чергою обмеженої місткості та механізмом пріоритезації оброблення.

2.3.1. Структурна модель потоків

Вхідний потік запитів $\Lambda(t)$ проходить через каскад фільтрів. Інтенсивність потоку, що спрямовується безпосередньо до черги нейромережевого аналізу, позначимо $\lambda_{queue}(t)$. Вона визначається як залишкова частина трафіку після застосування списків контролю доступу, що зберігаються в оперативній пам'яті (ACL):

$$\lambda_{queue}(t) = \Lambda(t) \cdot (1 - \eta_{ACL}(t))$$

де $\eta_{ACL}(t) \in [0,1]$ — ефективність фільтрації, що відображає частку трафіку, яка відсікається на ранньому етапі за рахунок чорного списку або, навпаки, пропускається без додаткової перевірки як завідомо довірена завдяки статичному білому списку (наприклад, для пошукових роботів). Водночас легітимний користувацький трафік з міркувань безпеки не додається до динамічного білого списку. Тому такі запити не можуть бути вилучені з подальшої перевірки і завжди формують навантаження λ_{queue}. У разі початку атаки мережеві адреси автономних агентів поступово ідентифікуються та заносяться до чорного списку. У результаті зростає значенняη_{ACL}, і шкідливий потік у значній частині відсікається ще до постановки в чергу нейромережевого аналізу з використанням моделі машинного навчання. Водночас потік легітимних запитів, що не може бути попередньо класифікований як довірений, продовжує надходити до контуру аналізу та визначає його базове навантаження.

2.3.2. Динаміка стану черги

Черга відкладеного аналізу $Q(t)$ описується рівнянням балансу потоків у часі, характерним для систем масового обслуговування (наближення рідинної моделі). Швидкість зміни довжини черги визначається різницею між інтенсивністю надходження нових запитів, що потребують аналізу, та сумарною продуктивністю модулів машинного аналізу. З урахуванням граничної умови щодо обмеженої місткості буфера ($0 \leq Q(t) \leq Q_{max}$), динаміка черги описується такою системою:

$$\frac{dQ(t)}{dt} = \begin{cases} 0, & \text{якщо } Q(t) = 0 \text{ і } \lambda_{queue}(t) \leq \mu_{total}(t) \\ 0, & \text{якщо } Q(t) = Q_{max} \text{ і } \lambda_{queue}(t) \geq \mu_{total}(t) \\ \lambda_{queue}(t) - \mu_{total}(t), & \text{в усіх інших випадках} \end{cases}$$

де

- $\lambda_{queue}(t)$ — інтенсивність потоку запитів, що надходять на етап відкладеного аналізу, зап./с;
- $\mu_{total}(t)$ — сумарна ефективна швидкість оброблення запитів модулем захисту, зап./с (детальний розрахунок наведено у п. 2.3.3);
- Q_{max} — максимальна місткість черги аналізатора, зап.

Якщо поточний розмір черги $Q(t)$ досягає граничного значення Q_{max}, система автоматично переходить у режим **Fail-Open**. У цьому стані нові надлишкові запити не додаються до черги аналізатора, а безпосередньо пропускаються до цільового вебдодатка без виконання глибокої перевірки. Такий підхід дозволяє уникнути блокування основного потоку оброблення та зберегти оброблення легітимного трафіку в умовах екстремальних перевантажень, хоча й за рахунок тимчасового зниження рівня фільтрації.

2.3.3. Математичний опис механізмів оптимізації

Для підвищення стійкості системи в пікових режимах передбачено два алгоритмічні механізми, які враховуються в подальшій математичній моделі.

МЕХАНІЗМ ВІДКЛАДЕНОЇ ВЕРИФІКАЦІЇ

Упродовж інтервалу T_{wait}, поки запит перебуває в черзі, мережеву адресу джерела може бути заблоковано іншим паралельним потоком. Наприклад, якщо автономний агент генерує десятки або сотні запитів за секунду, блокування може бути застосоване за результатом аналізу одного з попередніх запитів, тоді як інші запити продовжують накопичуватися в черзі. Тому безпосередньо перед передаванням запиту на вхід нейромережевого модуля виконується швидка перевірка актуального статусу IP-адреси.

Унаслідок цього частина запитів може бути відхилена без виконання ресурсоємного аналізу.

Для математичного опису цього механізму припустимо, що генерація шкідливих запитів автономним агентом утворює найпростіший потік подій, який підпорядковується закону розподілу Пуассона. Згідно з цим розподілом, імовірність того, що за час перебування поточного запиту в черзі (T_{wait}) від того ж самого джерела не надійде жодного іншого запиту, становить:

$$e^{-\lambda_{bot} \cdot T_{wait}(t)}$$

Відповідно, імовірність протилежної події, що за час очікування надійде хоча б один паралельний запит, який буде проаналізовано іншим обчислювальним потоком із подальшим блокуванням IP-адреси порушника, визначається як:

$$P_{skip}(t) = 1 - e^{-\lambda_{bot} \cdot T_{wait}(t)}$$

де

- $\lambda_{bot}(t) \approx \lambda_{queue}(t) - \lambda_{baseline}$ — оціночна інтенсивність бот-трафіку, зап./с, яка розраховується як різниця між поточним сплеском трафіку та історичним базовим рівнем;
- $T_{wait}(t)$ — поточний час очікування в черзі, с.

$$T_{wait}(t) = \frac{Q(t)}{\mu_{total}(t - \Delta t)}$$

де

- $Q(t)$ — поточна довжина черги, зап.;
- $\mu_{total}(t - \Delta t)$ — загальна продуктивність системи [зап./с], зафіксована на попередньому циклі вимірювання.

Це дає змогу ввести ефективну швидкість оброблення одного потоку аналізатора:

$$\mu_{eff}(t) = \frac{\mu_{worker}}{1 - P_{skip}(t)}$$

де μ_{worker} — номінальна швидкість обробки запитів одним обчислювальним потоком аналізатора, зап./с. Отже, сумарна швидкість оброблення черги системою становить:

$$\mu_{total}(t) = \min\left(N_{workers} \cdot \mu_{eff}(t), \mu_{cache}\right)$$

де μ_{cache} — максимальна фізична швидкодія читання з кешу відкладеної верифікації, зап./с. Додатково вводиться гранична умова для теоретичного випадку, коли весь вхідний трафік відсікається на рівні чорного списку. У такій ситуації частка відхилених запитів прямує до 1, тобто ймовірність відхилення запиту на подальший аналіз дорівнює 100%.

Отже можна зробити висновок, що зі зростанням інтенсивності атакувальної активності підвищується ймовірність відхилення запиту без повторного виконання перевірки, оскільки джерело порушення вже може бути ідентифіковане та заблоковане. У результаті формується ефект саморегуляції: за посилення атаки зменшується частка ресурсоємних перевірок, а навантаження на асинхронний контур обмежується за завдяки повторному використанню вже прийнятих рішень.

ЧАС РЕАКЦІЇ СИСТЕМИ

Критично важливим показником є час, необхідний системі для виявлення та блокування нового джерела загрози. Цей час визначається двома складовими. Перша складова — тривалість перебування запиту в черзі, яка обслуговується з ефективною сумарною швидкодією μ_{total}. Друга складова — час безпосереднього аналізу запиту в разі, якщо він не був відхилений на етапі швидкої перевірки та все ж надходить на поглиблене оброблення:

$$T_{reaction}(t) = T_{wait}(t) + T_{inference} = \frac{Q(t)}{\mu_{total}(t)} + \frac{1}{\mu_{worker}}$$

Перша складова, T_{wait}, зменшується завдяки механізму відкладеної верифікації, оскільки частина запитів відхиляється ще до виконання ресурсоємного аналізу, а ефективна швидкодія обслуговування черги зростає. Натомість друга складова, $T_{inference}$, у межах моделі розглядається як стала величина. Якщо запит уже передано на

нейромережевий модуль, його оброблення відбувається зі штатною швидкістю μ_{worker}, тобто без додаткових пришвидшувальних механізмів у критичній частині аналізу.

Умова успішного захисту (запобігання перевантаженню кластера під час атаки) зводиться до того, що час виявлення та блокування нового джерела загрози має бути меншим за характерний час, за який атакувальний потік здатен вичерпати доступний ресурс кластера:

$$T_{reaction} < T_{critical}$$

де

- $T_{reaction}$ — загальний час реакції системи безпеки на нову загрозу, с.;
- $T_{critical}$ — критичний час, с., протягом якого ще не заблокований автономний агент здатен вичерпати доступні вільні слоти кластера. У межах моделі він визначається параметрами граничної місткості системи (що задається через параметр K_{max}) та поточною інтенсивністю атакувального потоку λ_{bot}.

Математично критичний час вичерпання ресурсів виражається як відношення доступного вільного пулу з'єднань кластера до інтенсивності надходження шкідливих запитів. Оскільки гранична місткість системи визначається добутком максимально допустимої кількості серверів (K_{max}) на ємність одного вузла (C_{node}), а частина слотів завжди зайнята обслуговуванням легітимного фонового трафіку, формула набуває такого вигляду:

$$T_{critical} = \frac{K_{max} \cdot C_{node} - A_{baseline}}{\lambda_{bot}}$$

де

- K_{max} — адміністративне обмеження на максимальну кількість активних серверів у кластері;
- C_{node} — ємність одного обчислювального вузла (максимальна кількість одночасних запитів, п. 2.1.2);

- $A_{baseline}$ — базова кількість слотів пам'яті, зайнятих обслуговуванням легітимного трафіку на момент початку сплеску (розраховується в Ерлангах як $\lambda_{baseline} \cdot \tau$);
- λ_{bot} - інтенсивність генерації запитів автономним агентом, зап./с.

Відношення $T_{reaction} < T_{critical}$ демонструє взаємозв'язок між параметрами підсистеми захисту (місткість черги Q, швидкодія оброблення μ) та загальносистемними інфраструктурними обмеженнями кластера з урахуванням поточного корисного навантаження. На етапі проєктування системи вона дає можливість аналітично визначити, чи достатньо наявного вільного ресурсу кластера для утримання удару заданої потужності λ_{bot} до моменту остаточного блокування джерела атаки.

Примітка. Наведена вище формула описує теоретичну межу витривалості системи за умови миттєвої доступності всього резерву K_{max}. Однак на практиці хмарна інфраструктура характеризується інерційністю. Процес ініціалізації нових обчислювальних вузлів та їхнього введення в балансувальник потребує певного часу ($T_{provision}$) і цей час зазвичай становить від кількох десятків секунд до кількох хвилин залежно від типу віртуалізації. Якщо час реакції підсистеми захисту $T_{reaction}$ перевищує час вичерпання ресурсів поточної кількості активних серверів $K_{curr}(t)$, система тимчасово перейде в стан відмови **Fail-Close** до моменту розгортання нових вузлів. Тому для забезпечення безперервної доступності під час стрімких атак виникає додаткова жорстка умова для системи виявлення:

$$T_{reaction} < \frac{K_{curr}(t) \cdot C_{node} - A_{baseline}}{\lambda_{bot}}$$

де $K_{curr}(t)$ — фактична кількість серверів, активних на момент початку атаки ($K_{min} \leq K_{curr}(t) \leq K_{max}$).

Ця умова підкреслює критичну роль механізму відкладеної верифікації та швидкого відсікання трафіку. Підсистема захисту повинна ідентифікувати та заблокувати загрозу набагато швидше, ніж спрацює повільний контур автомасштабування ємності кластера, спираючись лише на наявний у цей момент ресурс.

2.3.4. Висновок до моделі

Запропонований метод асинхронного контуру захисту, що поєднує механізми попередньої ACL-фільтрації та відкладеної верифікації, математично обґрунтовує нелінійний характер зростання вимог до обчислювальних ресурсів під час кібератак. Завдяки ефекту саморегуляції результати глибокого аналізу перших шкідливих запитів миттєво поширюються на весь наступний потік від ідентифікованого джерела. Як наслідок, для успішної нейтралізації атаки з інтенсивністю λ_{bot} запитів за секунду необхідна сумарна обчислювальна потужність підсистеми захисту μ_{total} може бути на порядки меншою за інтенсивність вхідного шкідливого потоку ($\mu_{total} \ll \lambda_{bot}$). Це доводить спроможність запропонованої архітектури забезпечувати стійкість хмарного кластера до екстремальних перевантажень без необхідності симетричного нарощування дорогих обчислювальних потужностей, оскільки необхідний ресурс визначається виключно швидкістю появи нових унікальних джерел загроз, а не загальним обсягом генерованого ними трафіку.

2.4. Метод оцінювання ресурсної ефективності адаптивного кластера

Для кількісного оцінювання ефективності розробленої інформаційної технології необхідно виконати її порівняння з традиційною моделлю масового хостингу. Визначальною відмінністю між підходами є спосіб організації та використання обчислювальних ресурсів. У традиційній моделі множина сайтів розподіляється між фізичними серверами статично, а кожен сервер функціонує як відносно ізольована одиниця з локально закріпленими резервами ресурсу, що відповідає фрагментованій архітектурі. Натомість у запропонованій моделі ресурси окремих вузлів агрегуються у спільний динамічний пул, який керовано перерозподіляється відповідно до поточного навантаження та заданих вимог якості обслуговування.

2.4.1. Модель сценарію класичного хостингу

Нехай інфраструктура провайдера складається з S ізольованих серверів. Загальна множина вебресурсів $W = \{w_1, \ldots, w_n\}$ статично розподілена між цими серверами, так

що кожен сервер j обслуговує свою підмножину сайтів W_j. Для забезпечення виконання вимог SLA на кожному сервері j має виконуватися умова достатності оперативної пам'яті:

$$\sum_{w_i \in W_j} A_i(t) \cdot m_{unit} \leq RAM_{eff} \quad \forall t \in [0, T]$$

де

- $w_i \in W_j$ — ідентифікатор сайту, що розміщений на j-у сервері;
- $A_i(t) = \lambda_i(t) \cdot \tau(t)$ — інтенсивність вхідного навантаження в Ерлангах, є безрозмірною та чисельно відповідає очікуваній кількості одночасно активних запитів (зайнятих слотів пам'яті) на сервері;
- $\lambda_i(t)$ — інтенсивність запитів до i-го сайту в момент часу t;
- $\tau(t)$ — середній час утримання ресурсу (оброблення запиту) для всього агрегованого потоку, визначено в п. 2.1.4;
- m_{unit} — базовий квант пам'яті на один запит (визначено в п. 2.1);
- RAM_{eff} — ефективний обсяг пам'яті сервера, доступний для вебдодатків (повний обсяг мінус накладні витрати на ОС та системні процеси);
- $\forall t \in [0, T]$ — для будь-якого моменту часу t на проміжку T. Тобто перевантаження неприпустиме в жодний момент роботи.

Оскільки міграція вебресурсу між серверами характеризується високою інерційністю та, як правило, потребує перенесення даних, зміни конфігурацій і актуалізації DNS-записів, провайдер вимушений підтримувати резерв потужності на кожному сервері для компенсації локальних сплесків навантаження. Така практика призводить до двох типів втрат:

1. **Втрати від надлишкового резервування**. Для зменшення ризику перевантаження сервери, як правило, не завантажуються до межі 100. Натомість закладається запас ресурсу під пікові режими, зокрема формується резерв оперативної пам'яті (наприклад, на рівні 20—25% RAM) на кожному сервері. У

більшості часових інтервалів цей резерв не використовується за призначенням, що знижує фактичну ефективність консолідації ресурсу.

2. **Втрати від локального перевантаження**. Утримання значного резерву на кожному вузлі економічно невигідне, тому його обсяг зазвичай обмежується. Унаслідок цього на сервері j може виникати пікове навантаження, яке перевищує RAM_{eff}, і частина запитів відхиляється. Це може відбуватися навіть тоді, коли на інших серверах, зокрема на $j+1$, наявні вільні ресурси. Таким чином погіршується якість обслуговування на рівні конкретного вузла, а виконання показників SLA виявляється чутливішим до локальних пікових режимів.

Інтегральні витрати ресурсів для традиційної моделі F_{shared} визначаються як сума ресурсної ємності всіх серверів за звітний період T, незалежно від фактичного рівня їх завантаження. Оскільки в такій архітектурі кількість серверів стала (S), а окрема інфраструктура керування та балансування в межах фрагментованої архітектури відсутня ($RAM_{overhead} = 0$), інтегральний показник можна подати у спрощеному вигляді:

$$F_{shared} = S \cdot RAM_{total} \cdot T$$

де

- S — загальна кількість серверів, що обслуговують пул веб ресурсів;
- RAM_{total} — повний обсяг пам’яті сервера;
- T — тривалість звітного періоду.

Проблема. Навіть якщо на сервері j фактичне навантаження дорівнює нулю, ресурс RAM_j^{total} залишається зарезервованим і враховується як використаний у витратах, оскільки в межах фрагментованої архітектури він не може бути оперативно залучений для обслуговування сайтів з іншого сервера, що перебуває у стані перевантаження.

2.4.2. Модель сценарію динамічного кластера

У запропонованій архітектурі сервери об’єднані в єдиний кластер і розглядаються як спільний пул ресурсів. Відповідно обмеження задаються не для окремого фізичного сервера, а для кластера загалом. Поточне навантаження на кластер $\Lambda(t)$ визначається як сума навантажень усіх n вебресурсів у момент часу t, як це було задано в підрозділі 2.1.

Згідно із законом великих чисел і принципом статистичного мультиплексування, періоди підвищеної активності одних вебресурсів частково компенсуються періодами низької активності інших. Унаслідок цього пікові значення сумарного навантаження кластера, як правило, менші, ніж сума пікових навантажень, які виникали б у разі фрагментованого розміщення сайтів на ізольованих серверах традиційної моделі. Хоча ефект взаємного згладжування піків у певному обсязі проявляється і на окремому сервері shared-хостингу, на рівні всього кластера він суттєвіший через значно більшу кількість незалежних доменів, що формують агрегований потік [67].

Зі зростанням кількості сайтів n відносні коливання сумарного навантаження зменшуються, а розподіл навантаження в часі стає більш передбачуваним у статистичному сенсі. Це означає, що для забезпечення заданого рівня якості обслуговування доцільно формувати не множину локальних резервів на кожному сервері, а один спільний керований резерв на рівні кластера. Такий підхід дає можливість зменшити частку непродуктивного резервування ресурсу, оскільки резерв підтримується там, де він може бути використаний будь-яким доменом залежно від поточного стану системи. Отже, економічний ефект консолідації досягається насамперед завдяки різниці між сумою локальних резервів у фрагментованій архітектурі та обсягом єдиного глобального резерву в кластерній моделі.

Необхідна інтегральна ємність адаптивного кластера $F_{cluster}$, виражена в МіВ за період часу спостереження, наприклад тиждень, визначається як інтеграл від динамічно необхідного обсягу ресурсів за період T з урахуванням накладних витрат на балансування та керування:

$$F_{cluster} = \int_0^T \left(K_{req}(A_{total}(t)) \cdot RAM_{total} + RAM_{overhead}\right) dt$$

де

- $K_{req}(\Lambda_{total}(t))$ — функція розрахунку необхідної кількості серверів у момент часу t (визначається алгоритмом Ерланга з п. 2.2);
- RAM_{total} — повний обсяг пам'яті сервера;
- $RAM_{overhead}$ — пам'ять, необхідна на інфраструктуру керування та балансування;
- dt — крок інтегрування (інтервал часу).

2.4.3. Показник ефективності кластеризації

Економічна доцільність упровадження технології зумовлена усуненням фрагментації резерву, тобто ситуації, коли вільні ресурси одного сервера не можуть бути використані для компенсації дефіциту на іншому. Унаслідок цього зменшується потреба в локальному резервуванні та зростає частка ресурсу, що використовується за призначенням. Коефіцієнт ефективності консолідації $\xi_{consolidation}$ визначається як відношення інтегральних показників використання ресурсів:

$$\xi_{consolidation} = \left(1 - \frac{F_{cluster}}{F_{shared}}\right) \cdot 100\%$$

Цей показник відображає, на скільки відсотків менший обсяг фізичних ресурсів потрібен для обслуговування тієї самої кількості вебресурсів за збереження заданого рівня SLA, якщо перейти від фрагментованої архітектури до кластерної моделі з керованим спільним пулом. Показник $\xi_{consolidation}$ використовується для оцінювання саме ефекту консолідації та динамічного масштабування обчислювальних вузлів порівняно з класичною фрагментованою моделлю. Витрати ресурсів на асинхронний контур інтелектуального аналізу трафіку у значення $RAM_{overhead}$ та в інтегральний показник $F_{cluster}$ свідомо не включаються, оскільки у традиційному shared hosting такий контур зазвичай відсутній, а отже його врахування змішувало б порівняння архітектурної ефективності з додатковою функціональністю безпеки. Ресурсний бюджет

захисного контуру розглядається як окремий додатковий компонент і аналізується через модель стійкості та продуктивності підсистеми захисту (п. 2.3), не впливаючи на визначення $\xi_{consolidation}$. Отриманий резерв створює для провайдера дві принципові стратегії використання вивільнених потужностей:

- **Стратегія економії**. Зменшити кількість фізичного обладнання на $\xi_{consolidation}$ відсотків без погіршення поточного рівня надійності та якості обслуговування, визначеного SLA.
- **Стратегія підвищення якості**. Використати наявні фізичні потужності для досягнення вищих показників якості обслуговування. У кластерній архітектурі локальні сплески навантаження можуть бути поглинуті за рахунок спільного резерву і перерозподілу ресурсу, тоді як у традиційній схемі ті самі сплески частіше призводили б до відмов в обслуговуванні.

Отже, запропонований підхід забезпечує не лише зменшення витрат на інфраструктуру, а й підвищує керованість показників надійності. Це дає можливість розглядати рівень SLA як параметр, який може цілеспрямовано оптимізуватися разом із витратами ресурсу в межах єдиної моделі керування.

2.5. Висновки до другого розділу

У другому розділі розроблено математичне підґрунтя інформаційної технології адаптивного управління ресурсами хмарного кластера для динамічного хостингу веб-сервісів з урахуванням стохастичної моделі багатодоменного навантаження. На основі висновків першого розділу завдання формалізовано як задачу мінімізації інтегрального обсягу залучених обчислювальних ресурсів за умови дотримання ймовірнісних вимог SLA. Показано, що в середовищі масового хостингу доцільно розглядати оперативну пам’ять як критичний обмежувальний ресурс, а ємність кластера інтерпретувати через кількість одночасно підтримуваних каналів обслуговування.

Для кількісного опису якості обслуговування побудовано модель кластера у вигляді багатоканальної системи масового обслуговування з відмовами. Це дало змогу

пов’язати інтенсивність вхідного навантаження, середній час обслуговування запитів і доступну ємність кластера з імовірністю відмови в обслуговуванні. Як основний аналітичний інструмент обрано формулу Ерланга B, на базі якої сформовано обернену задачу визначення мінімально достатньої ємності кластера для забезпечення заданого рівня SLA. Для практичного використання запропоновано рекурентний метод обчислення ймовірностей відмови та алгоритм пошуку мінімальної конфігурації кластера, що задовольняє обмеження за якістю обслуговування.

У розділі також враховано динамічний характер реального хмарного середовища. Для запобігання коливанням контуру керування в модель введено механізм гістерезису, який усуває часті перемикання між режимами масштабування за незначних флуктуацій навантаження. Окремо формалізовано інерційність хмарної інфраструктури через час реакції контуру керування, що включає згладжування метрик, обчислення рішення та ініціалізацію нових вузлів. Це дозволило визначити умови, за яких система здатна компенсувати зростання навантаження без переходу в режим перевантаження, а також обґрунтувати необхідність архітектурного відокремлення швидких контурів реагування від повільного контуру автомасштабування.

Суттєвим результатом другого розділу є метод асинхронного захисту від прикладних атак L7. Підсистему подано як багатоканальну систему масового обслуговування з обмеженою чергою, де вхідний потік формується після попередньої ACL-фільтрації, а подальша обробка здійснюється модулями відкладеного інтелектуального аналізу. Показано, що за рахунок механізмів відкладеної верифікації і повторного використання раніше прийнятих рішень, підсистема здатна зменшувати навантаження на ресурсоємний контур аналізу навіть за інтенсивного шкідливого трафіку. У результаті обґрунтовано умову, за якої час реакції системи захисту має бути меншим за критичний час вичерпання доступного ресурсу кластера.

Окремо розроблено метод оцінки ресурсної ефективності адаптивного кластера порівняно з традиційною фрагментованою моделлю спільного хостингу. У цій постановці показано, що економічний ефект кластеризації досягається насамперед за

рахунок усунення локального резервування й заміни множини ізольованих резервів єдиним керованим ресурсним пулом. Запропонований коефіцієнт ефективності консолідації дозволяє кількісно оцінити виграш від переходу до кластерної архітектури з погляду зменшення необхідного обсягу фізичних ресурсів за збереження заданого рівня SLA.

Таким чином, у другому розділі розв’язано ключове завдання математичної формалізації адаптивного управління ресурсами хмарного кластера для динамічного хостингу. Отримано комплекс взаємопов’язаних моделей і методів, які описують визначення необхідної ємності кластера та умови динамічного масштабування, поведінку асинхронної підсистеми захисту та оцінку ефективності консолідації ресурсів. Сукупність цих результатів створює теоретичну основу для архітектурної реалізації запропонованої інформаційної технології в наступному розділі та для її подальшої експериментальної перевірки.

РОЗДІЛ 3. РОЗРОБКА ІНФОРМАЦІЙНОЇ ТЕХНОЛОГІЇ АДАПТИВНОГО ХОСТИНГУ

У другому розділі дисертаційної роботи розроблено комплекс математичних моделей, що описують процеси планування ємності кластера та фільтрації трафіку за стохастичного характеру навантаження. Формалізовано завдання мінімізації витрат ресурсів за умови дотримання заданого рівня SLA та обґрунтовано теоретичну ефективність переходу від фрагментованої архітектури до спільного пулу ресурсів.

Метою цього розділу є перетворення отриманих теоретичних залежностей на конкретні архітектурні рішення, алгоритмічні процедури та програмні компоненти. Основне завдання полягає у побудові інформаційної технології, що забезпечує функціонування балансувальника навантаження та керування динамічним хмарним кластером. Така технологія має реалізовувати обчислення необхідної кількості серверів $K_{req}(t)$ і керування чергами підсистеми захисту $Q(t)$ у режимі реального часу.

Обґрунтування вибору засобів реалізації

Критичні вимоги до системи, що безпосередньо випливають із математичної моделі (пп. 2.2-2.3), можна сформулювати так.

По-перше, необхідно мінімізувати накладні витрати. У виразі для оцінювання ефективності (п. 2.4) присутній параметр $F_{overhead}$, значення якого має бути якомога меншим, оскільки він безпосередньо знижує економічний ефект консолідації.

По-друге, потрібна передбачуваність затримок. Для коректної роботи механізму відкладеної верифікації та виконання умови $T_{reaction} < T_{critical}$ система не повинна демонструвати нерегламентовані паузи, зумовлені автоматичним керуванням пам’яттю. Такі паузи характерні для середовищ зі збирачем сміття і можуть проявлятися в реалізаціях на Java, Go або Python [87,88,89].

По-третє, важливо забезпечити безпечну роботу з пам’яттю. Оскільки балансувальник обробляє необроблений мережевий трафік на прикладному рівні, використання мов із ручним керуванням пам’яттю підвищує ризик класичних

уразливостей, зокрема виходу за межі буфера [24]. Для компонента, що входить до контуру захисту, такі ризики неприйнятні.

З огляду на зазначені обмеження, як основний інструмент для реалізації ядра системи обрано мову системного програмування Rust. Її модель керування пам’яттю, що ґрунтується на правилах володіння та позичання, гарантує безпеку роботи з пам’яттю без використання збирача сміття. Це дає змогу уникати нерегламентованих пауз, пов’язаних з автоматичним вивільненням пам’яті, і тим самим підтримувати передбачувані затримки в критичному шляху оброблення запитів. Додатковою перевагою є можливість ефективної реалізації асинхронного виконання з мінімальними накладними витратами. Це створює передумови для побудови висококонкурентного механізму оброблення запитів, який відповідає постановці багатоканальної системи масового обслуговування та дозволяє підтримувати значну кількість одночасних з’єднань за помірного споживання оперативної пам’яті. У сукупності зазначені властивості забезпечують відповідність ключовим вимогам моделі щодо мінімізації $F_{overhead}$ і дотримання часових обмежень, пов’язаних із реакцією системи на аномальну активність [90,91,92].

Як архітектурний принцип обрано підхід без збереження стану на обчислювальних вузлах (Stateless). Така організація є необхідною умовою для реалізації моделі динамічного пулу ресурсів (п. 2.4.2), оскільки дає змогу розглядати будь-який вузол як взаємозамінну одиницю ємності C_{node}. Як наслідок, спрощується керування масштабуванням, зменшується залежність від локальних особливостей вузлів і забезпечується можливість керованого перерозподілу навантаження в межах кластера.

3.1. Архітектура Stateless-кластера з розділеним шаром даних

Реалізація математичної моделі адаптивного керування потребує переходу від монолітної організації, у якій програмний код, дані та конфігурація зосереджені на одному вузлі, до розподіленої сервісно орієнтованої архітектури. У межах цього підходу обчислювальні компоненти та шар зберігання даних розглядаються як окремі

підсистеми з чітко визначеними межами відповідальності. Така декомпозиція створює умови для керованого масштабування і для узгодженого розподілу ресурсу в кластері.

Базовим архітектурним принципом обрано stateless-підхід, тобто відсутність збереження прикладного стану на обчислювальних вузлах. Це означає, що вузли не повинні містити довгоживучих даних, критичних для подальшого обслуговування запитів конкретного вебресурсу. Як наслідок, будь-який вузол може розглядатися як взаємозамінна одиниця обчислювальної ємності, що безпосередньо відповідає припущенням математичної моделі та забезпечує можливість оперативного перерозподілу навантаження в межах спільного пулу ресурсів.

3.1.1. Декомпозиція системи

Розроблена інформаційна технологія містить чотири ключові підсистеми, між якими організовано канали зв’язку для передавання запитів, телеметрії та керувальних сигналів. Узгоджена взаємодія всіх підсистем забезпечує адаптивне керування ресурсами та підтримання стійкості до загроз доступності. Узагальнену структурну схему технології подано на рис. 3.1.

Балансувальник — єдина точка входу до кластера, яка виконує функції централізованого приймання і первинного оброблення вхідного трафіку. Його робота організована в асинхронному режимі з орієнтацією на високу пропускну здатність і мінімальні затримки в критичному шляху. До основних функцій балансувальника належать: зберігання списків контролю доступу в оперативній пам’яті та виконання перевірки кожного запиту на наявність мережевої адреси джерела у відповідних списках; термінування TCP/TLS-з’єднань; передавання копії запиту до підсистеми поглибленої перевірки в асинхронному контурі; розподіл запитів між виконавцями на обчислювальних вузлах кластера.

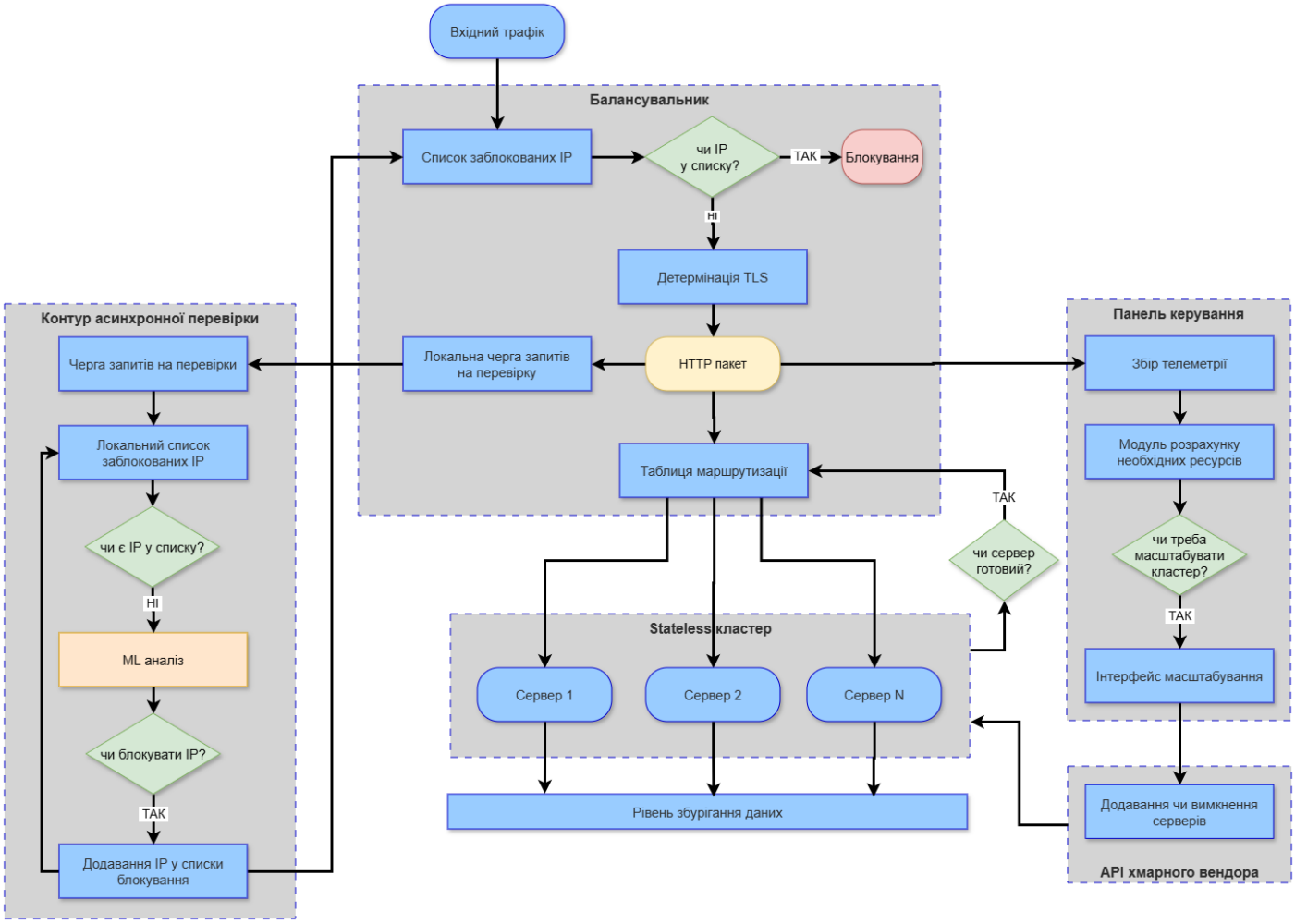

Рисунок 3.1 - Загальна структурна схема технології адаптивного кластера.

Панель керування — модуль, що забезпечує збирання телеметрії, обчислення необхідної ємності для дотримання показників якості та формування керувальних рішень щодо масштабування кластера. Саме в цьому модулі реалізовується логіка оцінювання стану черг $Q(t)$, а також підготовка вхідних параметрів для автоматичного масштабування на основі експлуатаційних метрик. Панель керування виконує роль керувального контуру, який реалізовує алгоритм зворотного зв'язку на основі ерлангівської постановки (п. 2.2), обчислює $K_{req}(t)$ і визначає необхідні дії щодо зміни кількості активних вузлів. Для забезпечення переносимості між хмарними середовищами та хмарними сервісами взаємодія з інфраструктурою реалізовується через абстрактний програмний шар керування ресурсами, без прив'язки до конкретних каналів

зв’язку з постачальниками хмарних сервісів. Цей шар надає уніфікований інтерфейс операцій масштабування кластера та інкапсулює відмінності конкретних платформ. Безпосередня інтеграція з інтерфейсами керування постачальників хмарних сервісів та хмарної інфраструктури реалізовується у вигляді окремих адаптерів, орієнтованих на відповідні середовища, зокрема AWS, Google Cloud, Microsoft Azure, OCI та DigitalOcean.

Stateless-кластер — динамічна група обчислювальних вузлів, на яких виконується прикладний код вебдодатків (PHP, Python, Node.js). Вузли функціонують у режимі без збереження стану, тобто не містять локальних даних, критичних для продовження обслуговування запитів конкретного ресурсу. Файлова система з програмним кодом сайтів під’єднується у режимі лише для читання з централізованого сховища. Така організація забезпечує тотожність версій коду на всіх вузлах кластера та зменшує ризик несанкціонованої модифікації скриптів безпосередньо на обчислювальних серверах, зокрема внаслідок шкідливого програмного забезпечення або шкідливого впливу.

Контур асинхронної перевірки — відокремлений сервіс, реалізований відповідно до патенту на корисну модель автора [35,30]. Його призначення полягає в обробленні запитів, що надходять до черги відкладеного аналізу, виконанні їхньої перевірки на ознаки шкідливої активності чи прикладних кіберзагроз шляхом застосування нейромережевого інтерфейсу та формуванні результату класифікації. Сформований вердикт передається балансувальнику і використовується для актуалізації списків контролю доступу, зокрема для оперативного оновлення правил фільтрації в синхронному тракті оброблення запитів.

Рівень зберігання даних — підсистема, що забезпечує централізоване розміщення програмного коду вебресурсів і зберігання динамічних даних, зокрема сесій, тимчасових файлів та баз даних. Така організація відокремлює обчислювальний шар від шару даних і підтримує stateless-режим роботи вузлів кластера.

3.1.2. Організація шару даних

Для забезпечення режиму без збереження стану всі дані винесено за межі обчислювального контуру до консолідованого шару даних, показаного в нижній частині рис. 3.1. Сесійні дані та кеш зберігаються окремо від програмного коду, що зменшує ризик локальних несанкціонованих змін і спрощує контроль цілісності середовища виконання. Відокремлення стану також усуває потребу синхронізувати файлові системи між обчислювальними вузлами та забезпечує можливість доступу до сесії користувача з будь-якого робочого вузла кластера.

Статичний контент і медіадані можуть зберігатися разом із кодом у випадку невеликих проєктів або виноситися до спільного мережевого сховища чи об’єктного сховища. Така організація створює передумови для подальшого розвитку, зокрема для інтеграції з мережами доставляння контенту та оптимізації віддачі статичних ресурсів.

Окремо слід зазначити, що винесення шару даних у спільне сховище не означає принципового збільшення обсягу даних або створення додаткових витрат порівняно з класичним shared-хостингом. Обсяг баз даних, медіафайлів і сесій визначається прикладними потребами вебресурсів і залишається співмірним незалежно від способу розміщення. Відмінність полягає не в кількості даних, а в організації доступу до них. У традиційній моделі ці дані можуть бути розподілені між окремими серверами, що ускладнює міграцію та масштабування. У запропонованій архітектурі той самий обсяг даних зосереджується у спільному шарі з керованим доступом, що забезпечує взаємозамінність обчислювальних вузлів і підтримує динамічний перерозподіл навантаження без дублювання даних і без необхідності локального резервування стану на кожному сервері.

3.1.3. Мережева топологія та безпека

Для забезпечення захисту інфраструктури від прямих атак реалізовано схему мережевої ізоляції з використанням приватних підмереж (VPC - Virtual Private Cloud) [93]. Схему розподілу зон доступу подано на рис. 3.2.

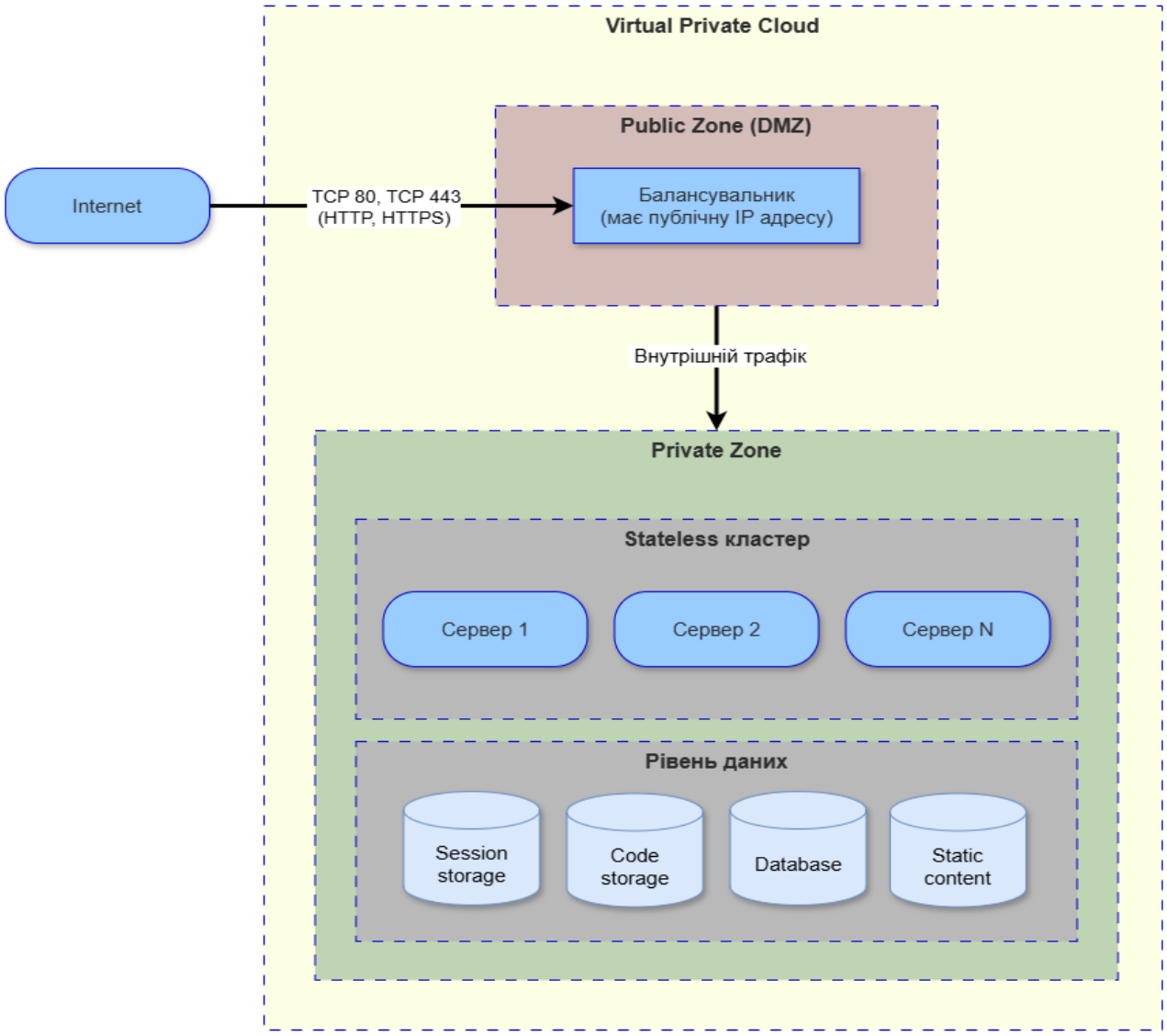

Рисунок 3.2 - Схема мережевої топології та ізоляції.

Публічна зона (Public Zone / DMZ). У цій зоні розміщуються лише екземпляри інтелектуального балансувальника навантаження. Це єдиний компонент архітектури, що експонує публічну сервісну IP-адресу та приймає вхідні з'єднання через порти 80/443. Доступ до нього регламентується жорсткими правилами міжмережевого екрана, що обмежують дозволені типи з'єднань і зменшують площу атаки для зовнішніх впливів. Для усунення одиничної точки відмови ILB розгортається щонайменше у двох екземплярах $N_{ILB} = 2$. Зовнішня сервісна адреса прив'язується до групи екземплярів, тоді як кожен екземпляр має власні технічні адреси для керування та обміну службовими даними. Перемикання зовнішньої адреси між екземплярами забезпечується типовим

інфраструктурним механізмом високої доступності. Зокрема, стандарт VRRP визначає правила вибору активного вузла для обслуговування віртуальної адреси та її автоматичного перенесення на резервний у разі відмови активного екземпляра [94]. Детальна реалізація механізму перемикання не входить до меж дослідження, однак факт резервування точки входу береться до уваги в подальших розрахунках як стала складова накладних витрат ресурсу. У розрахунках накладні витрати на точку входу беруться подвоєними відносно одного екземпляра, що підвищує точність оцінювання сумарної потреби в ресурсі кластера.

Приватна зона (Private Zone). У цій зоні розміщуються обчислювальні вузли та рівень даних. Вузли не мають публічних IP-адрес, а доступ до них здійснюється лише з боку балансувальника навантаження через внутрішню мережу з мінімальними затримками. Така організація виключає прямі з'єднання з Інтернету до серверів, що містять дані або виконують прикладний код. Як наслідок, усуваються типові вектори атак на інтерпретатори та сервісні інтерфейси керування. Зовнішній доступ до службових портів, зокрема для віддаленого адміністрування та доступу до баз даних, блокується на периметрі, а взаємодія дозволяється лише в межах внутрішнього сегмента.

3.1.4. Обґрунтування відсутності переконфігурації

Запропонована архітектура усуває інерційність керування, пов'язану зі зміною DNS, що була розглянута в п. 2.4.1. Оскільки DNS-запис домену вказує на сталу IP-адресу балансувальника навантаження, додавання або виведення з експлуатації робочих вузлів у приватній зоні відбувається прозоро для зовнішніх клієнтів. Унаслідок цього масштабування не потребує зміни DNS-записів і не залежить від часу їх розповсюдження в мережі [95].

Після того як підсистема керування ємністю повідомляє про готовність нового вузла, балансувальник актуалізує внутрішній перелік доступних робочих вузлів і правила маршрутизації. Таке оновлення виконується в робочому режимі та має малу

затримку, що уможливлює реакцію на зміну навантаження на рівні секунд. Це принципово відрізняється від традиційного підходу, у якому масштабування часто прив’язане до зміни DNS A-записів і може потребувати суттєво більшого часу через особливості кешування та розповсюдження DNS-відповідей.

3.2. Алгоритмічне забезпечення підсистеми маршрутизації та оброблення запитів

Ключовим компонентом розробленої інформаційної технології є інтелектуальний балансувальник навантаження (ILB). Він виконує роль єдиної точки входу до кластера та забезпечує термінування шифрованих з’єднань у середовищі масового хостингу, де одночасно обслуговуються тисячі незалежних доменів. Поширені рішення для балансування (зокрема Nginx, HAProxy та ін.) мають розвинені механізми конфігурації, розширення та динамічного керування. Водночас типові режими їх експлуатації та поширені підходи до оцінювання продуктивності, як правило, не розглядають балансувальник як активний елемент контуру забезпечення SLA за умов масового встановлення TLS-з’єднань, частого застосування змін конфігурації в робочому режимі та використання асинхронної аналітики трафіку. У межах запропонованої технології ILB інтегровано із зовнішнім шаром даних, який використовується для синхронізації списків контролю доступу, зокрема на основі швидкодійного доступу до правил у оперативній пам’яті. Окрім того, ILB реалізовує механізм асинхронного передавання параметрів вхідних запитів до черги поглибленого аналізу, що дає змогу відокремити ресурсоємні перевірки від критичного шляху оброблення легітимного трафіку [35]. Окремою складовою є абстрактний шар керування інфраструктурою, через який ILB може ініціювати масштабування обчислювальних вузлів шляхом взаємодії з інтерфейсами керування різних постачальників хмарної інфраструктури [61]. Така організація забезпечує перенесення технології між хмарними середовищами та зменшує залежність від конкретної платформи.

3.2.1. Модель асинхронного оброблення та конкурентності

Для забезпечення оброблення десятків тисяч одночасних з'єднань в межах одного обчислювального ядра застосовано модель кооперативної багатозадачності на основі бібліотеки Tokio. На відміну від підходу «потік на запит», за якого створення великої кількості потоків швидко вичерпує ресурси операційної системи та призводить до зростання накладних витрат, обрана архітектура ґрунтується на неблокувальному введенні-виведенні та подієвому керуванні з'єднаннями [24].

Мультиплексування мережевих подій реалізовується через системні механізми введення-виведення, зокрема epoll у Linux та kqueue у BSD. Це дозволяє підтримувати велику кількість активних з'єднань без пропорційного збільшення кількості потоків. У результаті оброблення виконується фіксованою кількістю потоків, яка узгоджується з кількістю фізичних ядер процесора, що забезпечує передбачувані накладні витрати та стабільну масштабованість точки входу за зростання кількості одночасних сесій.

3.2.2. Стратегія керування SSL-сертифікатами

Специфікою масового хостингу є необхідність підтримки SSL/TLS-шифрування для великої кількості доменів, які обслуговуються з однієї публічної IP-адреси [70]. За результатами оцінювання споживання оперативної пам'яті встановлено, що типовий розмір ланцюжка сертифікатів становить близько 2—5 KiB, що підтверджується як дослідженнями [96], так і практичними експериментами, описаними у розділі 4.4.1. Для кількісної ілюстрації надалі використовується масштаб $n = 10^4$ доменів як базовий розрахунковий сценарій. Він обраний для демонстрації порядку величин. Водночас цей вибір не є обмеженням моделі, оскільки витрати пам'яті на зберігання сертифікатів зростають майже лінійно із n і можуть бути оцінені як $RAM \approx n \cdot (2-5)KiB$.

За такого підходу для $n = 10^4$ доменів базова оцінка становить приблизно 20—50 MiB. З урахуванням накладних витрат на структури зберігання, індексацію, вирівнювання даних у пам'яті та службові метадані практичний обсяг може становити порядку 40—100 MiB. Для сучасних серверних платформ це несуттєво і не формує

критичного навантаження на дефіцитний ресурс. Якщо розглядати масштаб $n = 10^5$ доменів, відповідний обсяг зростає пропорційно і залишається на рівні близько сотень мегабайт, що також є прийнятним для типових конфігурацій оперативної пам’яті в серверному середовищі.

За цих умов доцільним є застосування стратегії повного попереднього завантаження сертифікатів в оперативну пам’ять. Вона усуває необхідність звернень до зовнішніх сховищ під час встановлення з’єднання та забезпечує мінімальну затримку вибору сертифіката, що є критичним для продуктивності точки входу в режимах масового встановлення TLS-сесій [49,21].

Реалізація сховища сертифікатів базується на таких принципах:

- **Структура даних**. Усі активні сертифікати зберігаються в оперативній пам’яті у вигляді хеш-таблиці HashMap<Domain, Arc>. Використання атомарного лічильника посилань (Arc) гарантує безпечне спільне використання одного й того самого об’єкта сертифіката в багатьох конкурентних контекстах виконання без дублювання даних в оперативній пам’яті [97]. Це зменшує накладні витрати та підтримує передбачувану місткість сховища сертифікатів за зростання кількості одночасних з’єднань.
- **Швидкодія**. Вибір необхідного сертифіката під час TLS Handshake виконується за очікуваний час $O(1)$ відносно кількості сертифікатів n , оскільки використовується доступ за ключем у хеш-таблиці, що розміщена в оперативній пам’яті [73]. Сертифікати попередньо завантажуються до пам’яті, тому в критичному шляху встановлення з’єднання відсутні операції читання з диска. Це зменшує варіативність затримок і робить час встановлення TLS-сесії більш передбачуваним у режимах масового підключення.
- **Динамічна актуалізація**. Операції додавання домену, видалення або оновлення сертифіката реалізовано як оброблення асинхронних подій. Фоновий процес моніторингу відстежує зміни у сховищі ключів і, у разі їх виявлення, виконує

атомарну вставку або заміну відповідного запису в хеш-таблиці. Завдяки засобам синхронізації Rust (зокрема RwLock або ArcSwap) актуалізація відбувається прозоро для активних з'єднань, не потребує перезапуску сервісу та не перериває встановлені сесії [97,98].

3.2.3. Інтеграція з підсистемою захисту

Для забезпечення максимальної швидкодії фільтрації, яка не залежить від мережевих затримок під час доступу до списків блокування, застосовано подійно-орієнтований механізм актуалізації кешу. Оновлення правил ініціюються подіями, а синхронний тракт обробляє запити, спираючись на локальні структури даних в оперативній пам'яті. Як комунікаційний механізм для розповсюдження подій використовується Redis Pub/Sub [99].

Алгоритм перевірки доступу та синхронізації організовано так, щоб критичний шлях оброблення запиту не залежав від мережевих звернень до зовнішніх сховищ і водночас забезпечував узгодженість правил у межах кластера.

1. **Механізм фільтрації**. Під час надходження запиту перевірка виконується виключно за локальним кешем в оперативній пам'яті (наприклад, `HashSet`). Якщо мережеву адресу джерела виявлено у списку блокування, запит відхиляється або застосовується механізм додаткової верифікації клієнта. Якщо адреси немає, запит передається далі до обчислювального контуру. На цьому етапі звернення до Redis не виконуються, що усуває мережеві затримки та не створює накладних витрат на кожному запиті.
2. **Механізм блокування**. У разі виявлення атакувальної активності асинхронний модуль аналізу формує керувальне рішення та публікує подію в канал `acl_updates`. Усі екземпляри балансувальника підписані на цей канал і після отримання повідомлення негайно додають відповідну адресу до локального списку блокування [99]. У такий спосіб узгоджене застосування правила досягається за

рахунок подійному доставленню оновлень, а затримка розповсюдження обмежується мілісекундним порядком величини.

3. **Синхронізація стану**. Під час холодного старту або після перезапуску екземпляр балансувальника завантажує актуальний знімок списків блокування з Redis одним запитом, синхронізуючи локальний кеш із поточним станом кластера. Після ініціалізації подальше оновлення стану виконується обробленням подій через підписку на канал оновлень.
4. **Автоматичне видалення зі списку блокування**. Локальний кеш підтримує механізм часу життя записів. Після завершення інтервалу блокування відповідний запис видаляється з оперативної пам’яті, відновлюючи доступ без додаткових мережевих звернень. Узгодженість підтримується тим, що на стороні Redis також використовується механізм автоматичного видалення за часом життя, тому локальна й централізована частини списку збігаються за логікою актуалізації [100].

3.2.4. Алгоритм динамічної реконфігурації топології

Функціонування модуля автоматичного масштабування ґрунтується на безперервному обчисленні необхідної кількості серверів $K_{req}(t)$ відповідно до ерлангівської моделі (п. 2.2). Оскільки балансувальник навантаження спостерігає агрегований вхідний потік $\Lambda(t)$ і має доступ до первинної телеметрії, саме він ініціює керувальні дії, пов’язані зі зміною складу обчислювальних вузлів, тобто з реконфігурацією топології кластера.

Алгоритм реконфігурації поділено на дві логічні гілки залежно від напряму масштабування. Кожна з них враховує інерційність операцій у хмарному середовищі, зокрема час створення або виведення вузла, а також вимогу безшовного обслуговування клієнтів. Такий поділ уможливлює кероване збільшення ємності в пікових режимах і коректне вивільнення ресурсів без ризику порушення показників якості обслуговування.

СЦЕНАРІЙ А: РОЗШИРЕННЯ КЛАСТЕРА.

Коли розрахункове значення $K_{req} > K_{current}$, система ініціює збільшення ємності кластера. Процедура розширення виконується послідовно та передбачає такі етапи:

1. **Ініціалізація додаткових вузлів**. Балансувальник формує керувальну подію та передає її до програмного шару керування хмарною інфраструктурою. Далі цей шар ініціює виклик API постачальника хмари для створення нових екземплярів робочих вузлів за заздалегідь визначеним шаблоном (образом), який містить необхідну конфігурацію середовища виконання.
2. **Очікування готовності**. Система очікує завершення завантаження операційної системи та запуску сервісів оброблення запитів на нових вузлах. Цей інтервал позначимо T_{boot}). До моменту підтвердження готовності вузол не включається до ротації, і трафік на нього не спрямовується.
3. **Перевірка стану та оновлення маршрутів**. Після підтвердження доступності сервера (наприклад, перевіркою відкритого порту 80 або контрольним запитом) балансувальник виконує оновлення маршрутизації. Застосовується підхід на основі принципу read-copy-update [98]: формується нова версія таблиці маршрутів, у яку додаються адреси нових робочих вузлів, після чого виконується атомарне перемикання на оновлену структуру. Це забезпечує прозоре введення вузла в експлуатацію без зупинки сервісу та без впливу на активні з’єднання, після чого новий вузол починає отримувати нові запити.

СЦЕНАРІЙ Б: ЗМЕНШЕННЯ КЛАСТЕРА.

Коли навантаження зменшується і виконується умова $K_{req} < K_{current}$, система має вивільнити надлишкові вузли без розриву активних з’єднань користувачів. Процедура зменшення ємності також виконується поетапно:

1. **Виведення вузла з ротації**. Балансувальник визначає вузол-кандидат на виведення та виконує оновлення таблиці маршрутизації за принципом read-copy-update [98]: формується нова версія списку доступних вузлів без адреси обраного

кандидата, після чого виконується атомарне перемикання на оновлену структуру. Починаючи з цього моменту нові запити на цей вузол не надходять.

2. **Завершення активних з’єднань**. Вузол продовжує обслуговувати вже встановлені з’єднання та запити, що перебувають у виконанні. Балансувальник відстежує завершення активності на вузлі за лічильниками поточних сесій або за механізмом обліку посилань на контекст обслуговування, що дає можливість визначити момент, коли вузол фактично звільнився від активних запитів.
3. **Фізичне вивільнення ресурсу**. Після того як кількість активних запитів на вузлі зменшується до нуля, або після спливу заданого максимального тайм-ауту для завершення, формується команда на виведення вузла з експлуатації на рівні хмарної інфраструктури, тобто на зупинку або знищення відповідного екземпляра віртуальної машини.

Зазначений підхід забезпечує безшовність масштабування та зменшує ризик деградації доступності під час вивільнення ресурсів. Для користувача момент виведення вузла прозорий, оскільки активні з’єднання завершуються коректно, а нові запити спрямовуються лише на вузли, що залишаються в ротації. Це дає можливість вивільняти ресурси провайдера автоматично, не порушуючи цільових параметрів якості обслуговування.

3.3. Програмна реалізація модуля масштабування

У попередніх підрозділах було розглянуто архітектуру оброблення запитів і базові механізми маршрутизації. Водночас наукова новизна роботи полягає не в самій процедурі розподілу трафіку, а в здатності системи автономно адаптувати ємність кластера до динамічних змін вхідного навантаження. Модуль предиктивного масштабування є програмною реалізацією математичної моделі планування ємності, обґрунтованої в підрозділі 2.2. Його призначенням є практична мінімізація цільової функції споживання ресурсів F (визначеної у п. 2.1.3) за умови дотримання обмежень якості обслуговування.

Процес керування полягає в безперервному перетворенні потоку телеметричних даних $\Lambda(t)$, який характеризує сумарну інтенсивність вхідних запитів, у розрахункове значення $K_{req}(t)$, тобто мінімально необхідну кількість активних серверів. У такий спосіб модуль виконує функцію замкненого контуру керування ємністю, у якому рішення про зміну топології кластера приймається на основі поточного стану системи та заданого рівня надійності.

Алгоритм визначення оптимальної конфігурації забезпечує виконання умови SLA:

$$P_{block} \leq P_{target}$$

де

- P_{block} — імовірність відмови в обслуговуванні через нестачу ресурсів,
- P_{target} — цільовий поріг надійності, визначений угодою про рівень обслуговування.

Принципово важливо, що в кожному циклі керування модуль не просто перевіряє виконання обмеження $P_{block} \leq P_{target}$, а визначає саме найменше значення $K_{req}(t)$, за якого це обмеження ще зберігається. Отже, програмна реалізація модуля масштабування фактично виконує покрокову мінімізацію поточної потреби в ресурсі за дотримання SLA. За інтегруванням у часі така стратегія відповідає мінімізації цільової функції F, введеної в п. 2.1.3, тобто мінімізації інтегральних витрат обчислювального ресурсу за умови збереження заданого рівня обслуговування.

3.3.1. Механізм збирання та агрегації телеметрії

Основою прийняття рішень модулем масштабування є наявність достовірних даних про сумарну інтенсивність вхідного потоку $\Lambda(t)$. В умовах асинхронної архітектури ILB, де тисячі запитів обробляються паралельно, збирання статистики ускладнюється потребою зберігати високу пропускну здатність і уникати блокувань у критичному шляху оброблення.

Застосування класичних примітивів взаємного виключення для захисту глобального лічильника запитів призвело б до конкуренції за спільний ресурс, що

супроводжується зростанням накладних витрат і зниженням продуктивності. У такому режимі оброблення телеметрії само по собі перетворюється на «вузьке місце» системи, оскільки кожен паралельний запит змушений очікувати доступу до спільного лічильника. Тому для забезпечення точності вимірювань без негативного впливу на пропускну здатність реалізовано дворівневу схему збирання та агрегації телеметрії:

1. **Рівень реєстрації подій**. На рівні обробників з'єднань застосовано неблокувальний підхід до синхронізації. Для підрахунку кількості запитів за одиницю часу та кількості активних з'єднань використано атомарні типи `std::sync::atomic::AtomicUsize` з ослабленим порядком доступу до пам'яті `Ordering::Relaxed` [101]. Це дає змогу інкрементувати лічильники без застосування взаємного вилучення й без зупинки інших потоків виконання, що принципово для висококонкурентних систем. За такого підходу витрати на реєстрацію подій залишаються малими і не перетворюються на «вузьке місце» в критичному шляху оброблення.
2. **Рівень агрегації даних**. Миттєве (поточне) оцінювання інтенсивності $\Lambda(t)$ є зашумленим сигналом із високою дисперсією, викликаною мікросплесками трафіку.Безпосереднє використання таких значень у формулі Ерланга призвело б до нестійкого керування та частих перемикань режимів масштабування. Тому для згладжування телеметрії застосовано агрегацію у ковзному часовому вікні. Для реалізації використано кільцевий буфер з N часових слотів, де кожен слот акумулює кількість подій за фіксований інтервал Δt (наприклад, 1 с). Згладжене значення інтенсивності $\Lambda_{smooth}(t)$, що передається на вхід алгоритму планування, визначається як середнє значення у вікні спостереження T_{window}:

$$\Lambda_{smooth}(t) = \frac{1}{T_{window}} \sum_{i=0}^{N-1} B_i \, , T_{window} = N \cdot \Delta t$$

де

- T_{window} — ширина вікна агрегації у секундах (конфігурований параметр, що визначає інерційність системи);
- B_i — кількість подій, зафіксована у i-му часовому слоті буфера.

Застосування ковзного вікна забезпечує надійніше оцінювання навантаження завдяки пригніченню короткочасних стохастичних коливань. Водночас зберігається чутливість до стійких змін інтенсивності трафіку, що необхідно для своєчасного прийняття керувальних рішень щодо масштабування.

3.3.2. Чисельна реалізація B-формули Ерланга

Програмна реалізація математичної моделі забезпечує перетворення оціненого вхідного навантаження $\Lambda(t)$ у розрахункову кількість фізичних серверів K_{req}. Архітектурно цей процес поділено на три послідовні етапи: визначення необхідної кількості абстрактних каналів обслуговування (M), перехід до фізичних одиниць ресурсу (K) та формування керувального сигналу (ΔK) для підсистеми масштабування.

ЕТАП 1. ЧИСЕЛЬНИЙ РОЗВ'ЯЗОК РІВНЯННЯ ЕРЛАНГА

Для визначення необхідної кількості каналів обслуговування M необхідно знайти корінь рівняння $B(\Lambda, M) \leq P_{target}$. Пряма імплементація класичної формули Ерланга (через факторіали $M!$) є неприпустимою в програмному коді через ризик переповнення розрядної сітки (Floating Point Overflow) та високу обчислювальну складність. Натомість, у програмному модулі реалізовано ітеративний алгоритм на базі рекурентного співвідношення [14,68]. На відміну від рекурсивних функцій, які можуть призвести до переповнення стека (Stack Overflow) за великих значень M, використано цикл з акумулятором стану [73].

Псевдокод алгоритму розрахунку має такий вигляд:

1. **Ініціалізація**. Задається початкове значення ймовірності блокування $P = 1.0$, що відповідає випадку нульової кількості каналів обслуговування ($m = 0$). Також визначається нижня межа пошуку $m = \lceil \Lambda \rceil$, оскільки пропускна здатність системи за кількістю каналів не може бути меншою за запропоноване навантаження.

2. **Ітерація**. У циклі виконується інкремент кількості каналів: $m \leftarrow m + 1$.
3. **Оновлення стану**. На кожному кроці обчислюється нове значення ймовірності блокування для поточного m на основі попереднього значення P_{prev}:

$$P_{new} = \frac{\Lambda \cdot P_{prev}}{m + \Lambda \cdot P_{prev}}$$

4. **Умова виходу**. Ітераційний процес зупиняється, коли виконується обмеження SLA, тобто $P_{new} \leq P_{target}$.

Запропонований підхід має лінійну часову складність $O(M)$ і є чисельно стійким. Усі проміжні значення P завжди розміщені в діапазоні $(0,1]$, що зменшує ризик числових переповнень і втрати точності під час обчислень у форматі IEEE `f64`.

ЕТАП 2. КОНВЕРТАЦІЯ У ФІЗИЧНІ РЕСУРСИ

Отримане значення необхідної кількості каналів M_{req} є логічною характеристикою місткості системи. Для формування керувального рішення щодо зміни складу кластера це значення перетворюється на кількість фізичних вузлів K_{calc}. Перетворення виконується з урахуванням параметра щільності C_{node}, який задає ємність одного обчислювального вузла та визначається у конфігурації системи:

$$K_{calc} = \left\lceil \frac{M_{req}}{C_{node}} \right\rceil$$

Такий підхід забезпечує апаратну незалежність алгоритму. У разі зміни типу серверів, зокрема під час переходу на вузли з іншим обсягом оперативної пам’яті, достатньо скоригувати значення C_{node} у конфігураційному файлі без модифікації програмного коду, що реалізовує математичну модель і алгоритм розрахунку.

ЕТАП 3. ФОРМУВАННЯ КЕРУВАЛЬНОГО СИГНАЛУ

Перед прийняттям керувального рішення розрахункове значення K_{calc} перевіряється на відповідність граничним умовам, визначеним у п. 2.1.4. Валідація виконується шляхом обмеження в інтервалі $[K_{min}, K_{max}]$:

$$K_{req} = \max(K_{min}, \min(K_{calc}, K_{max}))$$

де

- $K_{min} \geq 2$ — задає мінімальну кількість вузлів, необхідну для відмовостійкості навіть за відсутності навантаження;
- K_{max} — встановлює верхню межу масштабування для запобігання неконтрольованим витратам у разі аномальних сплесків трафіку, зокрема під час атак.

Нехай $K_{pending}$ — кількість вузлів, які вже ініційовано до запуску, але ще не введено в ротацію (очікування T_{boot}). Тоді ефективний запланований розмір кластера визначається як

$$K_{eff} = K_{current} + K_{pending}$$

На завершальному кроці визначається різниця між цільовою потребою K_{req} та поточним ефективним розміром кластера K_{eff}:

$$\Delta K = K_{req} - K_{eff}$$

Якщо $\Delta K > 0$, це означає дефіцит ресурсу, і формується сигнал `scale out`, тобто команда на додавання серверів. Ініціюється створення ΔK додаткових вузлів і збільшується лічильник $K_{pending}$. Після підтвердження готовності вузол переводиться з $K_{pending}$ у $K_{current}$ та додається до таблиці маршрутів.

Якщо $\Delta K < 0$, фіксується профіцит ресурсу, і генерується попередній сигнал `scale in` на зменшення ємності, який передається до модуля стабілізації для перевірки умов безпечного виведення вузлів.

3.3.3. Алгоритми стабілізації та гістерезис

Безпосереднє використання миттєвих результатів математичного розрахунку для керування інфраструктурою може спричиняти часті перемикання режимів масштабування, коли система багаторазово додає й виводить вузли у відповідь на незначні коливання трафіку. Такий режим неприйнятний, оскільки ініціалізація віртуальної машини має помітну інерційність (типово 30—90 с), а повторювані операції

створення та видалення екземплярів збільшують накладні витрати й підвищують ризик нестабільної роботи кластера [63].

Для усунення цього ефекту в програмному модулі реалізовано асиметричну стратегію керування, що поєднує пороговий і часовий гістерезис. Логіка свідомо робиться різною для розширення і зменшення кластера, оскільки ці режими мають різні ризики: дефіцит ресурсу безпосередньо загрожує порушенням SLA, тоді як профіцит ресурсу переважно впливає на економічну ефективність [63].

ШВИДКЕ РОЗШИРЕННЯ КЛАСТЕРА

Пріоритетом є збереження заданого рівня якості обслуговування, тому реакція на дефіцит ресурсу $\Delta K > 0$ негайна. Якщо розрахунок показує, що $K_{req} > K_{current}$, керувальна команда на додавання вузлів формується без затримок. Метою є мінімізація часу перебування системи в режимі підвищеного ризику, коли зростає ймовірність відмови в обслуговуванні. Зважаючи на інерційність підняття екземплярів (30—90 с) рішення про подальше розширення приймається з урахуванням уже ініційованих, але ще не активних вузлів. Для цього підтримується лічильник $K_{pending}$, а дефіцит ресурсу оцінюється відносно $K_{eff} = K_{current} + K_{pending}$. Такий підхід запобігає дублюванню керувальних команд у ситуації, коли нові вузли вже замовлені, але ще не беруть участі в обслуговуванні.

ПОВІЛЬНЕ ЗМЕНШЕННЯ КЛАСТЕРА

Рішення про виведення надлишкових вузлів $\Delta K < 0$ приймається лише після додаткової перевірки, яка знижує ймовірність передчасного вивільнення ресурсу та повторного швидкого повернення до розширення.

Етап А. Пороговий гістерезис. Перевіряється не лише факт виконання вимоги SLA, а й наявність запасу відносно цільового порога. Формально це задається як перевірка ймовірності блокування після потенційного зменшення кластера на необхідну кількість вузлів. Оскільки ерлангівська модель працює з кількістю каналів обслуговування, у перевірці використовується місткість у слотах:

$$M_{\Delta} = (K_{current} - \Delta K) \cdot C_{node}$$

Тоді умова допустимості зменшення набуває вигляду:

$$B(A(t), M_{\Delta}) \leq P_{target} \cdot (1 - \delta)$$

де

- $B(A(t), M_{\Delta})$ — формула Ерланга B;
- $A(t)$ — вхідне навантаження (величина, визначена в розділі 2.2);
- P_{target} — цільовий поріг імовірності блокування;
- δ — коефіцієнт запасу (конфігурований параметр, наприклад $\delta = 0{,}1$, що означає запас у 10%). Така умова гарантує, що після виведення одного вузла система зберігає запас стійкості і не переходить у критичний стан за найменшого зростання навантаження.

Етап Б. Часовий гістерезис. Навіть якщо порогова умова виконується, команда на фізичне виведення вузла блокується інтервалом стабілізації. Система фіксує момент останньої зміни топології T_{last_change}, а зменшення кластера дозволяється лише за умови:

$$T_{current} - T_{last_change} > T_{cooldown}$$

Параметр $T_{cooldown}$ задається конфігураційно. Типові значення перебувають у межах 5—10 хв. Це забезпечує, що масштабування «вниз» не відбувається одразу після короткочасного спадання трафіку, а рішення приймається лише після підтвердження стійкої тенденції.

У сукупності ці механізми стабілізують роботу кластера: система швидко реагує на зростання навантаження, щоб не допустити порушення SLA, і водночас зменшує ємність із затримкою та запасом, утримуючи ресурс для можливих повторних хвиль активності.

3.3.4. Абстракція інтерфейсу хмарного провайдера

Однією з ключових вимог до розробленої технології є відсутність жорсткої прив’язки до конкретного постачальника хмарних послуг. Пряма інтеграція клієнтських бібліотек керування інфраструктурою (наприклад, для AWS або Google Cloud)

безпосередньо в ядро системи призвела б до зростання складності, ускладнила б перенесення між платформами та зробила б архітектуру менш придатною до розширення. З цієї причини взаємодію з інфраструктурою реалізовано за шаблоном «адаптер»[102] із можливістю під'єднання реалізацій у вигляді окремих модулів.

Уніфікований контракт драйвера. Керувальний модуль системи взаємодіє з хмарним середовищем лише через абстрактний інтерфейс `CloudProvider`, який визначає мінімально необхідний набір операцій, незалежних від конкретної платформи. На рівні ядра система оперує узагальненими поняттями «екземпляр» і «мережева адреса», а деталі реалізації (тип ресурсу, спосіб автентифікації, формат ідентифікаторів тощо) інкапсулюються в адаптері:

- `spawn_instances(count: u32) -> Result<Vec<IpAddr>>` — створення нових обчислювальних вузлів;
- `terminate_instances(ids: Vec<IpAddr>) -> Result<()>` — виведення вузлів з експлуатації;
- `get_health_status() -> HealthReport` — перевірка доступності та працездатності інтерфейсу керування провайдера.

За такого підходу ядру системи неважливо, чи використовується AWS EC2, DigitalOcean Droplets, локальний кластер віртуалізації або приватна хмара. Усі відмінності зосереджуються в реалізації інтерфейсу, а сама така побудова архітектурно створює передумови для перенесення технології між різними інфраструктурними середовищами без зміни логіки керування ємністю.

Модульна архітектура драйверів. Для забезпечення можливості додавання нових платформ без модифікації логіки керування і без перекомпіляції ядра реалізації адаптерів винесено в окремі модулі, що можуть під'єднуватися динамічно під час запуску. Вибір активного адаптера і його параметризація виконуються через конфігураційний файл, наприклад `cloud_config.toml`:

```
[provider]
driver = "aws_driver"  # Або "digitalocean_plugin", "custom_script"
```

```
region = "eu-central-1"
credentials_path = "/etc/secrets/cloud_key.json"
```

Практичні наслідки такого підходу. По-перше, розширюваність забезпечується тим, що для додавання підтримки нового середовища достатньо реалізувати стандартний інтерфейс у вигляді окремого модуля, не втручаючись у механізми балансування та масштабування. По-друге, підтримується можливість тестування через імітаційний драйвер (умовний MockDriver), який відтворює створення та видалення екземплярів у пам’яті. Це дає змогу виконувати навантажувальні й функціональні експерименти без фактичних витрат на хмарні ресурси та без залежності від доступності зовнішніх API.

3.4. Програмна реалізація методу асинхронного захисту (L7)

У підрозділі 2.3 було побудовано математичну модель підсистеми захисту як багатоканальної системи масового обслуговування з чергою обмеженої місткості. Практична реалізація цієї моделі потребує зміни підходу до оброблення HTTP-запитів у порівнянні з традиційними міжмережевими екранами прикладного рівня. Класичні засоби захисту здебільшого працюють у синхронному режимі, коли кожен запит затримується до завершення його перевірки. За умов інтенсивного трафіку, зокрема під час прикладних атак, такий режим призводить до накопичення черг і зростання затримок, а підсистема безпеки перетворюється на обмежувальний елемент, який знижує пропускну здатність обслуговування легітимних користувачів [34,14].

Для усунення цього обмеження у розробленій системі застосовано асинхронну організацію захисту. Прийняття рішення щодо блокування винесено за межі критичного шляху оброблення запиту. Балансувальник навантаження виконує функцію точки входу та джерела телеметрії, тоді як ресурсоємні операції аналізу і класифікації для виявлення ознак прикладних кіберзагроз виконуються ізольованим модулем у відкладеному контурі. У такий спосіб забезпечується можливість поглиблена перевірка без блокування основного тракту обслуговування [35,30].

У цьому підрозділі розглядається програмна організація транспортного рівня та механізмів черг, що забезпечують передавання даних на відкладений аналіз, а також правила синхронізації результатів із балансувальником для актуалізації списків контролю доступу. Водночас алгоритм розпізнавання запитів, реалізований нейромережевою моделлю машинного навчання, розглядається як зовнішній компонент із наперед заданими характеристиками часу оброблення $T_{inference}$ [103,104]. Така постановка обґрунтована, оскільки питання побудови, навчання та оптимізації нейромережевих моделей виходять за межі цього дослідження і достатньо висвітлені в спеціалізованих наукових працях [103,104,105].

3.4.1. Архітектура асинхронного контуру

Для забезпечення повної ізоляції процесу оброблення клієнтських запитів від логіки їх аналізу застосовано підхід із розподіленою чергою. Ключовим принципом архітектури є винесення ресурсоємної аналітики в окремий контур і мінімізація дій балансувальника в критичному шляху. Тому на стороні балансувальника не виконується попереднє перетворення або поглиблене оброблення даних запиту. Його роль зводиться до формування телеметричних ознак і асинхронного передавання їх до черги відкладеного аналізу. Архітектурну схему асинхронного контуру подано на рис. 3.4.

Система складається з трьох взаємопов’язаних компонентів:

1. **Джерело подій — балансувальник (ILB)**. Балансувальник виконує роль джерела телеметричних подій і формує копію вхідного HTTP-запиту разом зі службовими заголовками та тілом повідомлення. Щоб не створювати блокування мережевого тракту під час передавання великих обсягів даних (зокрема для POST/PUT-запитів із вкладеннями), застосовано механізм усічення корисного навантаження. Якщо розмір тіла запиту перевищує встановлений ліміт L_{max} (наприклад, 4 KiB), до черги передається лише початковий фрагмент тіла та метадані про повний розмір (параметр Content-Length) [86]. Такий підхід дає змогу аналізувати заголовки та характерні ознаки атак, що зазвичай проявляються на початку запиту, не

перевантажуючи транспортний рівень передаванням медіафайлів або інших великих даних.

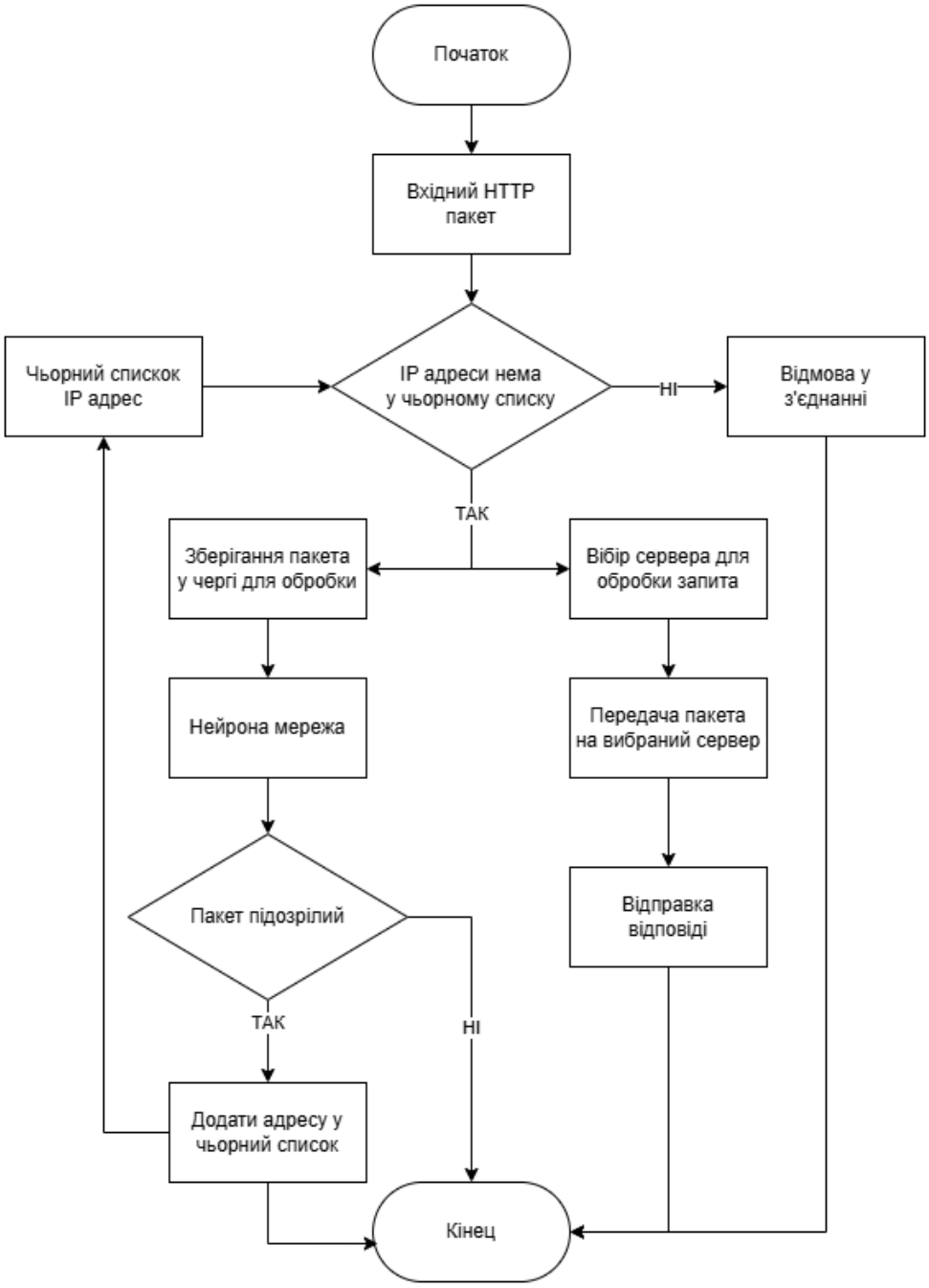

Рисунок 3.4 - Архітектурна схема асинхронного контуру аналізу трафіку [24].

2. **Транспортний шар — брокер повідомлень**. Для буферизації сирих запитів використовується зовнішній брокер повідомлень (наприклад, RabbitMQ або Apache Kafka). Це забезпечує розчеплення функцій між трактом обслуговування трафіку та

контуром аналізу. Унаслідок цього зміна моделі нейромережевого аналізу або логіки підготовки даних не потребує модифікації коду балансувальника. Окрім того, брокер виконує роль буфера, здатного поглинати короткочасні сплески інтенсивності, що тимчасово перевищують пропускну здатність аналізаторів, і тим самим зменшує ризик втрати подій [14]. У роботі як референсну реалізацію брокера повідомлень використано RabbitMQ, однак це не є обмеженням запропонованої технології. Критичними є не конкретний продукт, а властивості транспортного шару: неблокувальна публікація без очікування підтверджень, можливість роботи з недовговічними повідомленнями та підтримка політик обмеження/очищення черги. Для інших брокерів застосовуються функціонально еквівалентні механізми.

3. **Обробник пакетів — ML Worker**. Це окремий обчислювальний компонент, що підписується на чергу сирих даних і виконує повний цикл підготовки та аналізу. До його функцій належать парсинг HTTP-заголовків і нормалізація адресного рядка, формування часових та поведінкових ознак на основі історії звернень, перетворення даних у нормалізований вектор ознак і передавання цього вектора до нейромережевої моделі [103,104]. У разі виявлення атакувальної активності модуль формує керувальний сигнал блокування та передає його назад до ILB для актуалізації списків контролю доступу [35].

Запропонована архітектура забезпечує мінімальний вплив аналізу на критичний шлях обслуговування. Балансувальник залишається максимально легковагим, а обчислювальна складність зосереджується в ізольованому асинхронному контурі. Навіть у разі часткової деградації або повної зупинки підсистеми аналізу балансувальник зберігає здатність пропускати легітимний трафік. Водночас у такому режимі знижується рівень захищеності, що відповідає обраній політиці пріоритезації доступності сервісу.

3.4.2. Реалізація неблокувального транспорту

Критичною вимогою до підсистеми транспортування даних є мінімізація її впливу на основний тракт оброблення запитів. Затримки, перевантаження або збої на стороні брокера повідомлень не повинні ставати джерелом деградації пропускної здатності балансувальника. Тому архітектуру підсистеми передавання телеметрії спроєктовано за принципом пріоритезації доступності: балансувальник має зберігати стабільну роботу навіть ціною втрати частини аналітичних повідомлень. Для цього використано стратегію «відправив і не очікую відповіді» з примусовим скиданням повідомлень у разі перевантаження [65]. За такого підходу стійкість точки входу розглядається як пріоритетна властивість системи, тоді як гарантії доставлення аналітичних даних свідомо не є визначальними.

Для ізоляції головного циклу оброблення подій від мережевих затримок реалізовано асинхронний шаблон передавання повідомлень через внутрішній буфер обмеженої місткості. Балансувальник використовує канал у оперативній пам’яті з фіксованим розміром. Додавання повідомлення до цього каналу виконується неблокувальним викликом. Якщо канал заповнений, що є індикатором проблем на транспортному рівні або на стороні брокера, повідомлення відхиляється негайно ще до етапу серіалізації. Це зменшує накладні витрати і запобігає витратам процесорного часу на формування даних, які все одно не можуть бути доставлені. Вичитування з каналу та мережеве передавання повідомлень виконує окремий фоновий потік, що дає можливість повністю відокремити критичний шлях оброблення запиту від операцій введення-виведення.

Взаємодію з брокером повідомлень (на прикладі RabbitMQ) налаштовано на максимальну пропускну здатність із мінімізацією затримок:

- **No Confirms**. Підтвердження доставлення вимкнено, тобто балансувальник не очікує відповідей від брокера та не блокується під час публікації повідомлень [106].

- **Non-Persistent**. Повідомлення маркуються як такі, що не потребують запису на диск, щоб уникати затримок введення-виведення на стороні брокера. Втрата частини повідомлень у разі перезапуску не є критичною, оскільки вони мають телеметричний характер і швидко втрачають актуальність [107].

Щоб запобігти накопиченню застарілих повідомлень у черзі брокера, що могло б призводити до оброблення неактуальних даних і помилкових рішень захисту, на рівні RabbitMQ застосовано політики обмеження та ротації:

- **x-max-length**. Обмеження максимальної довжини черги.
- **x-overflow = drop-head**. У разі переповнення черги видаляються найстаріші повідомлення, звільняючи місце для нових, актуальних подій [108].
- **Message TTL**. Встановлюється час життя повідомлення (наприклад, 30 с). Якщо обробник не отримав повідомлення в межах цього інтервалу, воно вважається неактуальним і видаляється автоматично [109].

Зазначена комбінація механізмів спрямована на те, щоб навіть повна деградація або відмова підсистеми аналізу не спричиняла сповільнення роботи балансувальника і не погіршувала показники SLA для легітимного трафіку. Це узгоджується з вимогою, сформульованою в постановці завдання: механізми захисту не повинні впливати на якість обслуговування в критичному шляху.

Компонент-обробник (ML Worker) функціонує як незалежний сервіс, що реалізовує конвеєрне оброблення потоку сирих запитів. Його призначення полягає у підготовці даних у формі, придатній для ефективного використання обчислювальних ресурсів модуля аналізу, без втручання в логіку прийняття керувальних рішень на рівні балансувальника. У такий спосіб складність нормалізації, побудови ознак та організації викликів інференсу зосереджується в асинхронному контурі, а критичний шлях оброблення запитів залишається мінімальним.

Операція інференсу має сталу складову накладних витрат, пов’язану з ініціалізацією контексту виконання, підготовкою буферів та запуском обчислення. Цей ефект особливо виражений у разі використання графічних пришвидшувачів. Тому

поштучне оброблення запитів знижує ефективність використання ресурсу та обмежує пропускну здатність контуру аналізу. Для усунення цього обмеження реалізовано механізм пакетного оброблення з динамічним формуванням пакетів [110].

ML Worker вичитує з черги брокера множину повідомлень розміром до N_{batch}, де N_{batch} — конфігураційний параметр. Сформований пакет перетворюється на узгоджений масив ознак і передається до модуля аналізу як єдина структура даних. Це дає змогу ефективніше використовувати векторні інструкції процесора та паралельні обчислювальні ресурси, зокрема при прискоренні на GPU [110]. У підсумку пропускна здатність асинхронного контуру аналізу зростає у порівнянні з послідовним поелементним обробленням, що безпосередньо підвищує швидкість виявлення й блокування нових джерел аномальної активності.

З огляду на асинхронний характер черги можливі ситуації, коли під час очікування повідомлення в RabbitMQ рішення щодо джерела трафіку вже змінюється. Зокрема, поки запит від атакувального агента перебуває в черзі, відповідну IP-адресу може бути ідентифіковано та заблоковано іншим паралельним потоком аналізу. У такому разі повторне виконання інференсу для вже відомого порушника зайве і створює непродуктивні витрати ресурсу.

Для усунення повторних обчислень у ML Worker реалізовано проміжний етап перевірки. Перед передаванням сформованого пакета на інференс виконується швидка перевірка IP-адрес запитів за локальним кешем блокувань. Запити, що походять від адрес, які вже мають статус блокування, вилучаються з пакета без виконання ресурсоємного аналізу. Такий механізм дає можливість оперативно гасити залишкові хвилі атаки і зменшувати навантаження на нейромережевий модуль завдяки повторному використанню вже прийнятих рішень.

Логіка побудови ознак, виявлення аномалій та класифікації трафіку інкапсульована в ізольованому модулі аналізу. У межах цього дослідження нейромережевий компонент розглядається як зовнішній елемент із заданими характеристиками продуктивності. Система взаємодіє з ним через абстрактний

інтерфейс, який приймає «сирий» HTTP пакет і повертає вердикт, що може бути безпосередньо використаний для формування керувальної дії, зокрема для оновлення правил доступу.

Запропонована абстракція забезпечує модульність системи, оскільки транспортний контур і логіка аналізу ізольовані на рівні інтерфейсу. Заміна механізму детекції, зокрема оновлення параметрів моделі або використання іншої реалізації нейромережевого модуля, не потребує змін у компоненті балансувальника та не впливає на його критичний шлях оброблення запитів. Унаслідок цього оновлення аналітичного компонента можуть виконуватися незалежно від транспортного шару, що спрощує супровід і дає змогу еволюційно вдосконалювати систему.

Окрім того, такий підхід зменшує вимоги до оперативної пам’яті у воркері. Обробник не зобов’язаний зберігати розширений контекст оригінальних запитів протягом усього циклу обчислень. Після отримання масиву `verdicts` воркер виконує послідовний перегляд результатів і, у разі перевищення порога $score > threshold$, формує керувальну дію блокування для відповідної IP-адреси. Це дає можливість швидко застосовувати рішення та обмежувати обсяг даних, що утримуються в пам’яті під час роботи асинхронного контуру аналізу.

3.4.3. Емуляція затримок та стратегії відмов

Критичною характеристикою підсистеми захисту є її поведінка в умовах перевантаження, коли інтенсивність надходження подій для аналізу перевищує фізичну пропускну здатність обчислювальних модулів. У запропонованій архітектурі обрано стратегію **Fail-Open** (пропуск трафіку в разі деградації аналітичного контуру) [77]. Якщо черга повідомлень переповнюється через недоступність ML Worker або через його недостатню продуктивність, балансувальник припиняє буферизацію нових запитів для аналізу, однак продовжує передавати трафік до обчислювальних вузлів. Такий режим допомагає зберегти доступність сервісу та уникнути ситуації, коли сама підсистема безпеки стає обмежувальним елементом.

Альтернативна стратегія **Fail-Closed** (блокування всього неперевіреного трафіку) за умов прикладних атак ризикована [77]. У цьому разі затримки або відмова в роботі аналізаторів можуть призвести до блокування легітимних запитів і фактично сформувати самоспричинену відмову в обслуговуванні [65]. Тому в цій роботі пріоритет надається стабільності критичного шляху оброблення запитів, а аналітичний контур розглядається як допоміжний механізм, який не має права знижувати рівень SLA через власні затримки.

У разі атаки, пропускання неперевіреного трафіку тимчасово збільшує навантаження на обчислювальні вузли. У цей момент активується контур автоматичного масштабування, описаний у підрозділі 3.3. Фіксуючи зростання агрегованого потоку запитів, алгоритм планування ємності ініціює додавання вузлів кластера. Отже, система в режимі деградації аналітики переходить від переважно фільтраційної поведінки до режиму поглинання навантаження, зберігаючи доступність сервісу ціною підвищення інфраструктурних витрат у короткостроковому інтервалі.

Оскільки навчання та налаштування нейромережевої моделі не входять до меж цього дослідження, у розділі 4 використано модуль-імітатор аналітичного контуру. Його призначенням є відтворення затримки оброблення без розгортання повноцінної ML-інфраструктури, щоб окремо дослідити вплив часу аналізу на динаміку черг і реакцію системи. Затримка інференсу моделюється нормальною випадковою величиною

$$T_{inference} = \mathcal{N}(\mu_{delay}, \sigma^2)$$

де

- μ_{delay} — середня затримка;
- σ — параметр розсіювання.

У моделі взято $\mu_{delay} \approx 45$ мс. Це значення використано як інженерну оцінку, узгоджену з результатами [104], де для легковагової моделі LUCID на ресурсно-обмеженій платформі показано продуктивність близько 23 тис. зразків за секунду при розмірі пакету 1024, що відповідає приблизно 44,5 мс на пакет. Тому вибране значення

є реалістичним за порядком величини для експериментального моделювання взаємодії аналітичного контуру з чергою повідомлень і механізмом масштабування.

3.5. Висновки до третього розділу

У цьому розділі подано архітектурні та алгоритмічні рішення ключових компонентів адаптивної хмарної системи, математичні моделі якої обґрунтовано в розділі 2. За результатами архітектурного проєктування та програмної реалізації дослідницького прототипу встановлено такі основні положення.

1. **Архітектурно обґрунтовано** побудову дослідницького прототипу у формі stateless-кластера. Показано доцільність переходу від монолітної організації до сервісно орієнтованої архітектури з розмежуванням зон відповідальності між точкою входу, обчислювальним шаром і шаром даних. Застосування монтування коду в режимі лише для читання та мережева ізоляція приватної зони зменшують ризик несанкціонованих змін прикладного коду й обмежують доступ до критичних сервісів із зовнішньої мережі.
2. **Визначено склад і принципи реалізації** дослідницького прототипу високопродуктивного балансувальника навантаження прикладного рівня. На основі асинхронної моделі введення-виведення передбачено модуль оброблення з’єднань, придатний для підтримання великої кількості конкурентних TLS-сесій за помірних накладних витрат. Для зменшення варіативності затримок передбачено зберігання сертифікатів в оперативній пам’яті, а для оновлення маршрутів і конфігурації — підхід із атомарним перемиканням версій структур даних без простою.
3. **Виконано програмну реалізацію алгоритмічного ядра** модуля масштабування на основі математичної моделі планування ємності, обґрунтованої в розділі 2. Для забезпечення чисельної стійкості використано ітераційний рекурентний алгоритм обчислення ймовірності блокування, який не потребує факторіалів, уникає переповнення та має лінійну складність $O(M)$. Для стабілізації керування передбачено

механізми порогового і часового гістерезису, що зменшують ризик частих перемикань режимів масштабування за короткочасних коливань навантаження.

4. **Передбачено засоби інтеграції** з асинхронним контуром прикладного захисту. Запропоновано механізм подійної синхронізації списків блокування через Redis Pub/Sub, що дає можливість оперативно застосовувати правила фільтрації в синхронному тракті без мережевих звернень у критичному шляху оброблення запитів.
5. **Спроєктовано абстрактний інтерфейс керування ресурсами** та модульну організацію адаптерів до API постачальників хмарної інфраструктури. Це створює передумови для розгортання запропонованої технології в різних хмарних і гібридних середовищах без жорсткої прив’язки до конкретної платформи.

Отже, у розділі сформовано архітектурні та алгоритмічні засади й визначено склад програмних компонентів, необхідних для побудови дослідницького прототипу та подальшої експериментальної перевірки його працездатності й ефективності. Експериментальну перевірку в характерних сценаріях масового багатодоменного хостингу подано в розділі 4.

РОЗДІЛ 4. ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ІНФОРМАЦІЙНОЇ ТЕХНОЛОГІЇ

Метою цього розділу є експериментальна перевірка працездатності розробленої інформаційної технології, аналіз програмної реалізації її основних складових та підтвердження практичної придатності математичних моделей, запропонованих у розділах 2 і 3. Цей розділ не обмежується описом програмного стенда або технічних деталей реалізації. Його призначення полягає у верифікації основних положень дисертаційного дослідження в умовах, наближених до реальної експлуатації багатодоменного хостингового середовища.

У межах розділу перевіряються три ключові положення. По-перше, багатодоменний балансувальник здатний стабільно обслуговувати велику кількість доменів у динамічному середовищі із захищеними з’єднаннями. По-друге, модуль прогнозування навантаження та масштабування кластера забезпечує керовану адаптацію кількості обчислювальних вузлів до зміни інтенсивності потоку запитів. По-третє, винесення контуру аналізу підозрілого трафіку в асинхронну підсистему підвищує стійкість системи до атак типу HTTP flood і не створює критичного навантаження на основний контур оброблення запитів.

Для оцінювання роботи системи використано сукупність показників, що відображають як продуктивність, так і ресурсну ефективність запропонованого підходу. Основними з них є інтенсивність вхідного потоку запитів, середній час відповіді, квантіль затримки p95, частка помилок, кількість втрачених запитів, рівень завантаження процесора, обсяг використаної оперативної пам’яті, довжина локальних і зовнішніх черг, кількість активних серверів у пулі, тривалість масштабування та час стабілізації після зміни режиму роботи. Для сценаріїв із захищеними з’єднаннями додатково враховано кількість завантажених сертифікатів і доменів, а для дослідження захисного контуру — час до блокування джерела атаки, кількість заблокованих адрес та вплив порога накопичення подій на швидкість стабілізації системи. Окрему увагу

приділено показнику RSS, який використовується для оцінювання фактичного обсягу оперативної пам’яті, зайнятої процесом балансувальника.

Експериментальне дослідження виконано на спеціально підготовленому стенді, який забезпечив контрольованість умов проведення випробувань і відтворюваність одержаних результатів. Детальний опис складу стенда, параметрів середовища та використаних інструментів наведено в підрозділі 4.2.

Важливою особливістю експериментального стенда є використання не спрощеної, а наближеної до практики багатодоменної схеми захищених з’єднань. У дослідженні застосовано великий набір окремих сертифікатів для тестових доменів, що дало змогу відтворити реальні умови роботи балансувальника в середовищі масового вебхостингу. Завдяки цьому експерименти враховують не лише логіку маршрутизації запитів, а й фактичний вплив великої кількості доменів на використання оперативної пам’яті та на загальну продуктивність системи. Детальну побудову сертифікатного середовища наведено в підрозділі 4.2.

Логіка побудови розділу підпорядкована послідовній перевірці запропонованої інформаційної технології. Спочатку подається опис програмної реалізації та експериментального стенда. Далі перевіряється достатність стенда і відсутність сторонніх «вузьких місць», які могли б спотворити результати подальших досліджень. Після цього розглядаються експерименти, присвячені багатодоменному режиму роботи балансувальника, перевірці модуля розрахунку ємності та масштабування, а також порівнянню ресурсної ефективності класичної моделі shared hosting і адаптивного кластерного підходу. Завершується розділ дослідженням поведінки системи під атакою та оцінюванням внеску асинхронного контуру захисту в підвищення її стійкості.

4.1. Програмна реалізація інформаційної технології

Для практичної реалізації запропонованої інформаційної технології створено програмний комплекс, який відтворює основні процеси функціонування хмарного середовища динамічного хостингу веб-сервісів. Його побудовано так, щоб забезпечити

приймання та маршрутизацію клієнтських запитів, підтримку великої кількості доменів із захищеними з’єднаннями, динамічне керування складом пулу серверів прикладного рівня, збирання телеметричних даних, а також асинхронний аналіз підозрілих запитів із подальшим застосуванням рішень захисного контуру. На відміну від спрощених моделей, у яких перевіряються лише окремі фрагменти функціональності, розроблений комплекс охоплює повний цикл оброблення запиту в багатодоменному середовищі та створює основу для подальшої експериментальної перевірки моделей, запропонованих у попередніх розділах.

Програмна реалізація інформаційної технології складається з кількох взаємопов’язаних компонентів, зображених на контекстній архітектурній діаграмі (рис. 4.1). Центральним елементом є інтелектуальний балансувальник навантаження ILB, який виконує функції фронтального вузла системи. Окремо реалізовано процес збирання та передавання телеметричних даних Ingester, емулятор нейромережевого аналізу ML-Worker, емулятор серверів прикладного рівня Backend, а також набір автоматизованих сценаріїв запуску і прогону експериментів. Такий поділ дав змогу, з одного боку, зберегти архітектурну цілісність системи, а з другого, забезпечити ізольоване керування окремими контурами під час експериментів.

Балансувальник ILB є ядром розробленої системи. Саме він приймає клієнтські з’єднання, завершує сеанси захищеного обміну, перевіряє дозволеність джерела, обирає потрібний сертифікат домену, виконує маршрутизацію на сервери прикладного рівня та повертає відповідь клієнтові. Одночасно з цим він формує копії запитів для асинхронного аналізу і здійснює збирання власних телеметричних показників. Така організація дає змогу поєднати в одному програмному вузлі функції багатодоменного балансування, оперативного керування пулом серверів і первинного включення запитів у захисний контур.

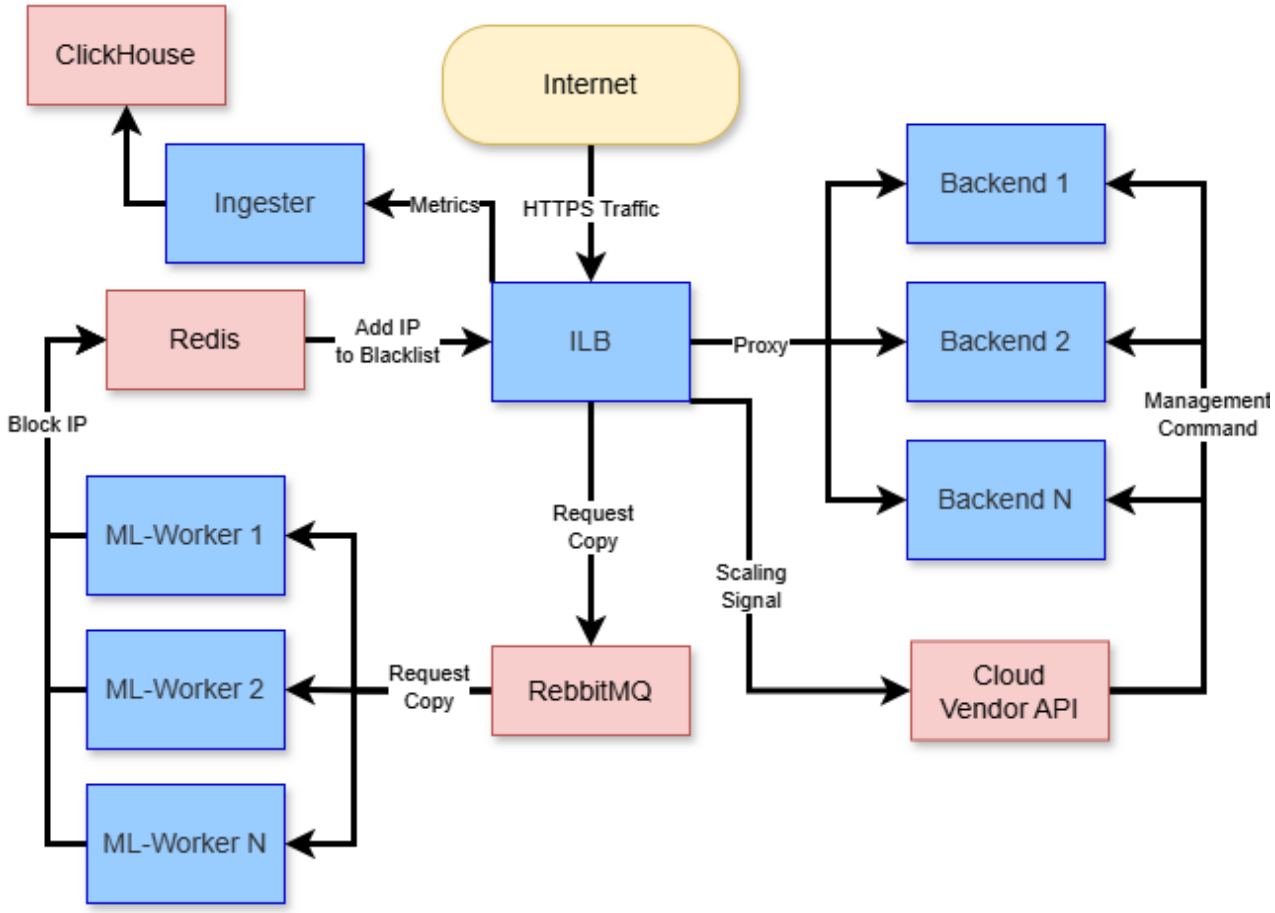

Рисунок 4.1 - Контекстна діаграма архітектури інформаційної технології.

Початкове налаштування системи забезпечує модуль `config` (рис. 4.2), який виконує читання, десеріалізацію та перевірку конфігураційних параметрів. У ньому задаються мережеві параметри, склад пулу серверів, характеристики телеметрії, параметри асинхронного передавання запитів і правила роботи допоміжних контурів. Модуль `server` реалізовує фронтальну частину балансувальника, приймає клієнтські з'єднання та забезпечує основну логіку проходження запиту через систему. Модуль `tls` відповідає за завантаження доменних сертифікатів, вибір належного сертифіката за ім'ям домену та динамічне оновлення сертифікатної бази без повної зупинки процесу. Саме цей модуль принципово важливий для моделювання середовища масового багатодоменного хостингу, де склад доменів і сертифікатів не є сталим.

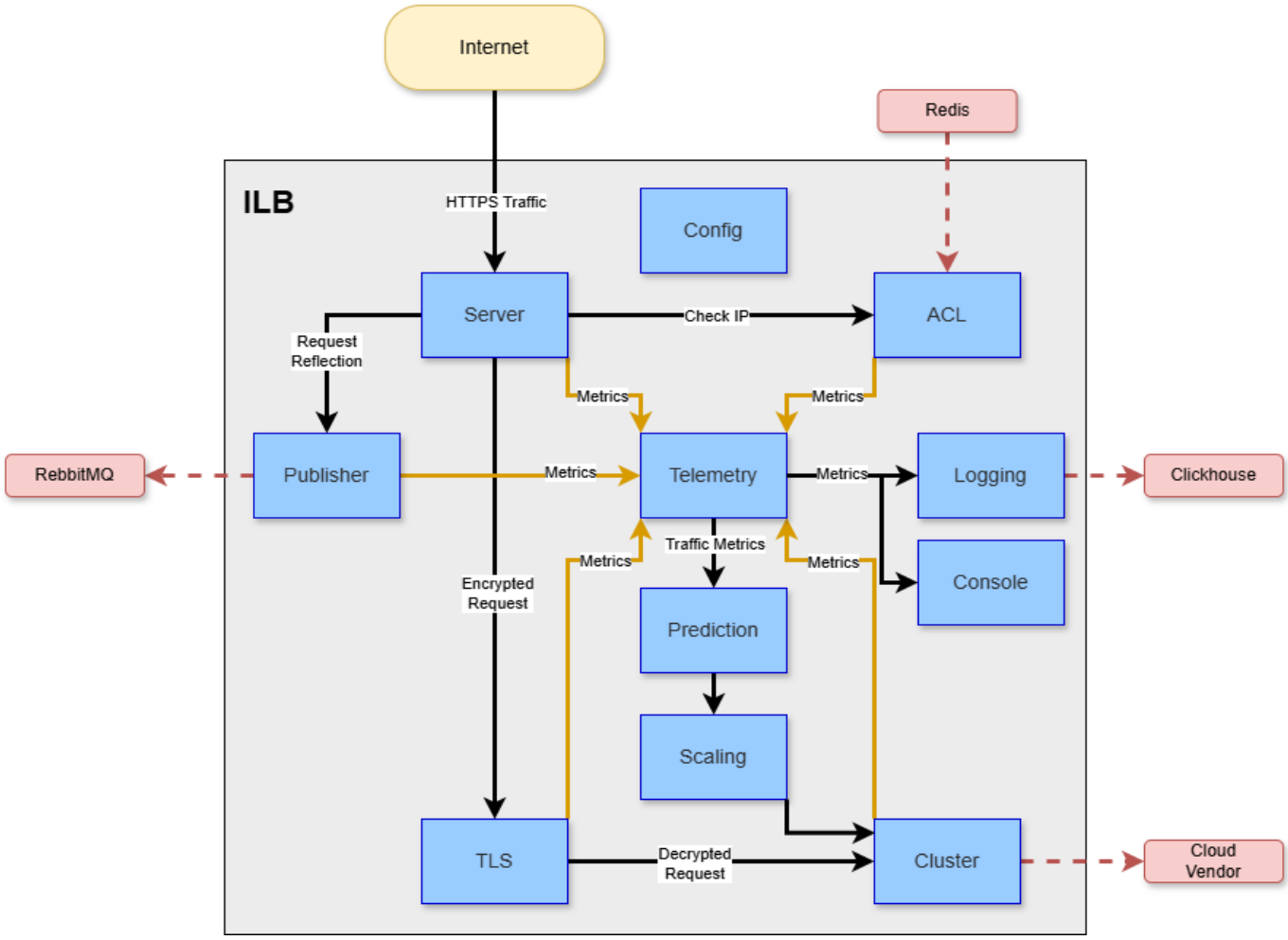

Рисунок 4.2 - Модульна діаграма ILB.

Окремий контур керування ємністю реалізовано через модулі `prediction` і `scaling`. Модуль `prediction` оцінює необхідну кількість серверів прикладного рівня на основі згладженої інтенсивності потоку запитів, цільового рівня обслуговування та обмежень, що випливають із математичної моделі втрат. У його роботі враховано не лише поточний стан навантаження, а й період охолодження між рішеннями, що зменшує ймовірність коливань складу кластера. Програмну реалізацію ключових елементів логіки модуля наведено в додатку 1. Модуль `scaling` виконує прикладну частину цього рішення. Він керує введенням нових вузлів у пул, задає затримку їх готовності до приймання навантаження та організовує контрольоване виведення вузлів із кластера при спаді інтенсивності трафіку. Програмну реалізацію ключових елементів логіки модуля наведено в додатку 2. У сукупності ці два модулі забезпечують практичне відтворення

механізму адаптивного масштабування, який було обґрунтовано в теоретичній частині дисертації.

Модуль `cluster` зберігає поточний стан серверів прикладного рівня, забезпечує вибір вузла для нового запиту, веде облік активних звернень і підтримує режими `active`, `draining` та `disabled`. Така модель станів дає можливість не лише рівномірно розподіляти навантаження, а й коректно вводити та виводити вузли з обслуговування без різких стрибків у роботі системи. Програмну реалізацію ключових елементів логіки модуля наведено в додатку 3.

Перевірка клієнтських адрес до початку маршрутизації виконується модулем `acl`, який підтримує локальний список блокування та отримує його оновлення через Redis. Завдяки цьому рішення, отримані в асинхронному контурі аналізу, можуть бути швидко перенесені в робочий контур балансувальника без необхідності перебудови всієї конфігурації. Програмну реалізацію ключових елементів логіки модуля наведено в додатку 4.

Передавання копій HTTP-запитів до контуру асинхронного аналізу здійснює модуль `publisher`. У ньому реалізовано локальну чергу, контроль переповнення, обмеження часу життя повідомлень та фонове передавання даних у RabbitMQ. Таке рішення дає змогу не блокувати основний контур обслуговування клієнта додатковими діями, пов’язаними з аналізом підозрілої активності. Фактично запит спочатку проходить швидкий синхронний контур оброблення, тоді як його копія незалежно передається до фонової аналітичної підсистеми. Саме це розділення є однією з ключових ознак запропонованої архітектури.

Для спостереження за станом системи та накопичення результатів роботи в ILB реалізовано окремий телеметричний контур. Модуль `telemetry` формує щосекундні телеметричні зрізи, у яких агрегуються основні показники функціонування балансувальника, пулу серверів і захисного контуру. Модуль `logging` записує ці дані до локальних файлів у структурованому вигляді. Таке рішення принципове, оскільки допомагає відокремити збирання телеметрії від її подальшого аналітичного

опрацювання та не створювати прямих звернень до зовнішньої бази даних у критичному контурі оброблення запитів. Для локального спостереження під час експериментів використовується модуль `console`, який відображає поточний стан системи та короткі службові події без засмічення основного потоку виводу.

Допоміжні програмні компоненти виконують функції, необхідні для замикання повного технологічного циклу. Процес Ingester зчитує файли телеметрії, сформовані балансувальником, об’єднує записи в пакети та передає їх до ClickHouse для централізованого зберігання і подальшого аналізу. Це дає змогу виносити важкі операції накопичення та оброблення телеметрії за межі основного програмного вузла. Компонент ML-Worker виконує роль емулятора нейромережевого аналізу. Він приймає копії запитів із черги повідомлень, імітує часові витрати на аналіз, формує рішення щодо підозрілої активності та через Redis передає оновлення до модуля списку блокування. Завдяки цьому в експериментальному середовищі відтворюється не лише наявність аналітичного контуру, а й його часовий вплив на поведінку всієї системи.

Емулятор Backend використовується для формування керованого пулу серверів прикладного рівня з наперед заданими характеристиками відповіді. Його застосування дає можливість змінювати кількість доступних вузлів, параметри часу оброблення запиту, обсяг сформованої відповіді та інші величини, не залучаючи сторонні програмні платформи, поведінка яких могла б ускладнити інтерпретацію результатів. Саме завдяки цьому забезпечується експериментальна чистота дослідження, коли реакція системи аналізується в умовах контрольованого, а не випадкового серверного середовища. Набір автоматизованих сценаріїв запуску використовується для підготовки конфігурацій, керування серіями експериментів, запуску навантаження, ротації параметрів та збирання результатів, що підвищує відтворюваність усіх наступних досліджень.

Розроблена програмна реалізація інформаційної технології являє собою цілісний багатокомпонентний комплекс, у якому розмежовано контур обслуговування клієнтських запитів, контур керування ємністю, контур асинхронного аналізу та контур телеметричного спостереження. Така архітектура дає змогу дослідити не лише окремі

властивості балансувальника, а й узгоджену роботу всіх елементів системи в умовах багатодоменного навантаження, масштабування та захисту від атак на прикладному рівні. Параметри розгортання цього комплексу та характеристику експериментального середовища наведено в підрозділі 4.2.

4.2. Опис експериментального стенду та параметрів середовища дослідження

Для проведення експериментального дослідження було створено спеціалізований стенд, призначений для відтворення умов функціонування багатодоменного хмарного середовища. Стенд розгорнуто на орендованих віртуальних серверах із виділеними обчислювальними ресурсами. Кожен вузол працював під керуванням Ubuntu 24.04 LTS. Використання саме серверів із виділеними віртуальними процесорами дало змогу зменшити варіативність результатів, пов’язану з конкуренцією за процесорний час у спільному середовищі віртуалізації, і тим самим підвищити відтворюваність експериментальних вимірювань.

Експериментальний стенд складався з чотирьох серверів (рис. 4.3), кожен з яких виконував окрему функцію в загальній архітектурі дослідження. Перший вузол, ILB Server, використовувався для запуску інтелектуального балансувальника навантаження та локального процесу збирання телеметричних показників. Саме на цьому сервері приймався вхідний трафік, виконувалась термінація захищених з’єднань, маршрутизація запитів до серверів прикладного рівня, а також формувалися дані для подальшого аналізу роботи системи. Другий вузол, Backend Server, призначався для запуску кількох екземплярів емулятора серверної частини вебзастосунку. Кожен екземпляр працював на окремому порту, що дозволяло формувати пул backend-серверів із контрольованою та передбачуваною продуктивністю. Третій вузол, Worker Server, використовувався для запуску емулятора нейромережевого аналізу, а також допоміжних інфраструктурних компонентів, зокрема Redis, RabbitMQ та ClickHouse. Четвертий вузол, Loader Server, виконував роль генератора навантаження. На ньому запускалися сценарії тестування,

побудовані на базі k6, причому саме цей вузол було обрано найбільш продуктивним, щоб зменшити ризик того, що джерелом обмеження стане не досліджувана система, а засіб формування трафіку.

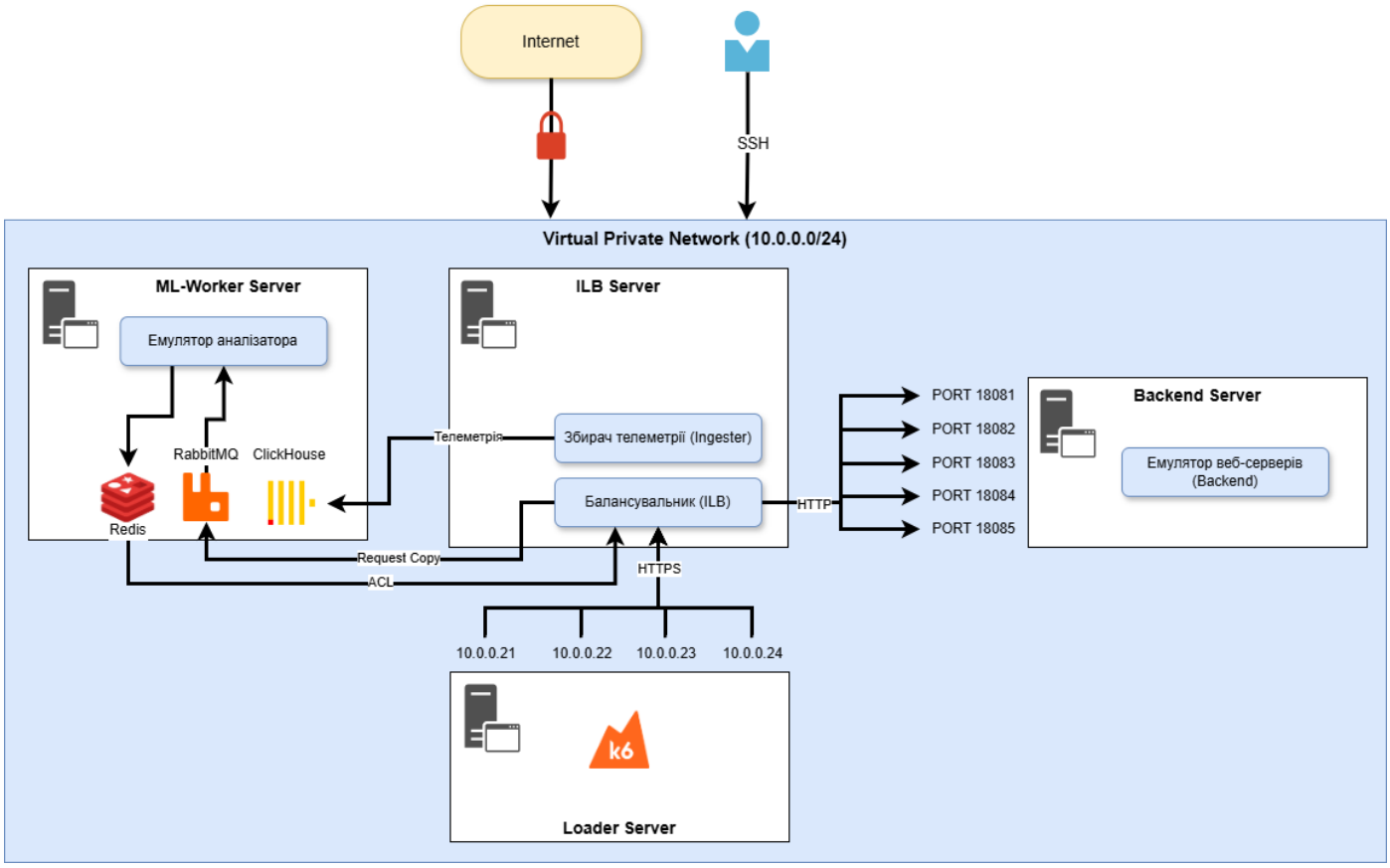

Рисунок 4.3 - Топологія експериментального стенду.

У межах стенда всі сервери були розміщені в одному датацентрі та поєднані внутрішньою віртуальною локальною мережею. Доступ із глобальної мережі був заборонений для всіх вузлів, окрім доступу з IP-адреси автора, що дало змогу усунути сторонній трафік і зменшити вплив зовнішніх чинників на результати експериментів. Така конфігурація дала змогу зосередити вимірювання саме на поведінці досліджуваної інформаційної технології, а не на нестабільності відкритого мережевого середовища.

Особливу увагу під час побудови стенда було приділено забезпеченню можливості генерування інтенсивного навантаження протягом тривалого часу. На Loader Server було додано п’ять локальних IP-адрес, що допомогло зменшити ризик вичерпання діапазону клієнтських TCP-портів під час формування великої кількості паралельних з’єднань. На

всіх серверах було розширено діапазон доступних локальних портів до значень від 1000 до 65535. Перед запуском прикладних процесів у кожній активній сесії додатково встановлювались підвищені ліміти на кількість відкритих файлів і процесів за допомогою команд `ulimit -n 65535` та `ulimit -u 65535`. Це дало можливість уникнути впливу стандартних системних обмежень на результати тестування в режимах інтенсивного навантаження.

Для побудови сценаріїв навантаження та подальшого оброблення результатів використано декілька інструментів. Основним засобом генерації трафіку в усіх ключових експериментах став k6, який застосовувався для відтворення контрольованого HTTP- та HTTPS-навантаження, зокрема в режимі великої кількості доменів і тривалих серій вимірювань. Для побудови добових і тижневих профілів навантаження, масштабування трафіку та подальшої агрегації результатів використовувались Python і бібліотеки pandas та numpy. Така комбінація інструментів дала змогу поєднати відтворення реалістичних профілів попиту з можливістю автоматизованого аналізу великих масивів експериментальних даних.

У складі стенда було також реалізовано підсистему телеметричного спостереження. Балансувальник записував щосекундні телеметричні знімки до локальних файлів. Паралельно з ним працював окремий процес збирання телеметрії, який зчитував ці файли, формував пакети даних і передавав їх до ClickHouse для подальшого аналізу. Такий підхід допоміг відокремити швидкий контур оброблення запитів від контуру накопичення й аналітичного опрацювання даних, не покладаючи додаткове навантаження безпосередньо на балансувальник у момент обслуговування клієнтських з’єднань.

Окремою складовою експериментального середовища стала підсистема захищених з’єднань, побудована за ієрархічною схемою, наближеною до реальної практики публічної інфраструктури сертифікації. У тестовому середовищі було використано самопідписаний кореневий центр сертифікації на основі ECDSA P-384, проміжний центр сертифікації також на основі ECDSA P-384 та 10 000 окремих

серверних сертифікатів для доменів вигляду site000001.example … site010000.example, сформованих на основі ECDSA P-256. Кожен серверний сертифікат мав власну пару ключів, унікальний серійний номер, індивідуальні поля Subject і Subject Alternative Name, а також типовий набір службових розширень, необхідних для коректного використання в ланцюжку сертифікації. Сертифікати зберігалися у форматі PEM, а для кожного домену на боці балансувальника використовувався готовий серверний пакет, що містив серверний і проміжний сертифікати в правильному порядку подання.

Застосування саме такого підходу було принциповим для коректності експериментів. У спрощеній моделі, де використовувався б один умовний сертифікат або невеликий їх набір, вплив підсистеми захищених з’єднань на використання оперативної пам’яті та процесор балансувальника був би суттєво заниженим. У запропонованому стенді, навпаки, відтворювалися умови, близькі до багатодоменного хостингового середовища, у якому балансувальник повинен одночасно зберігати в пам’яті великий набір TLS-ідентичностей, виконувати вибір сертифіката за ім’ям домену та підтримувати велику кількість віртуальних хостів. Це дало змогу підвищити достовірність оцінювання впливу кількості доменів на витрати оперативної пам’яті, процесора та на межі продуктивності системи.

У підсумку, побудований експериментальний стенд поєднує ізольоване мережеве середовище, контрольовану генерацію навантаження, передбачуваний пул backend-вузлів, окремий контур асинхронного аналізу та засоби централізованого збирання телеметрії. Така конфігурація створює достатні умови для проведення відтворюваних експериментів, спрямованих на перевірку працездатності розробленої інформаційної технології, оцінювання її масштабованості, ресурсної ефективності та стійкості до атак на прикладному рівні.

4.3. Верифікація експериментального стенду

Перед переходом до дослідження, власне, запропонованої інформаційної технології необхідно було перевірити, що результати подальших експериментів

визначатимуться властивостями балансувальника та пов’язаних із ним модулів, а не обмеженнями допоміжної інфраструктури. З цією метою виконано верифікацію експериментального стенда за трьома напрямами: оцінено можливості сервера генерації навантаження, перевірено продуктивність пулу backend-емуляторів та проаналізовано достатність мережевої інфраструктури.

Для проведення перевірки використано окремий сценарій навантаження `k6/backend-rps-test.js`, наведений у додатку 6. Як сервер прикладного рівня застосовано спеціалізований емулятор, що повертає HTML-сторінку обсягом 50 KiB та підтримує фіксований час відповіді на рівні однієї секунди. Такий підхід дає можливість працювати з наперед відомою продуктивністю backend-рівня і унеможливити вплив стороннього програмного забезпечення на результати експерименту. Емулятори запускалися на окремому сервері з двома віртуальними процесорами та 16 GiB оперативної пам’яті. З метою визначення граничної пропускної здатності системи було деактивовано програмні обмеження щодо кількості одночасних запитів на рівні серверної частини (backend).

Навантаження формувалося засобом k6, запущеним на окремому сервері з чотирма віртуальними процесорами та 16 GiB оперативної пам’яті. У межах дослідження послідовно перевірялися конфігурації з 5, 10 та 15 екземплярами backend-емулятора. Для кожної конфігурації проводився прогін у діапазоні від 4000 до 7000 запитів за секунду. Основною метою цього етапу було встановити практичну верхню межу навантаження, після якої саме генератор трафіку починає втрачати здатність стабільно підтримувати заданий режим. Додатково було необхідно підтвердити, що робочий рівень навантаження для подальших досліджень гарантовано лежить нижче цієї межі.

Результати пошуку граничного режиму показали, що в зоні до приблизно 5800 RPS стенд загалом зберігає працездатність, а подальше зростання інтенсивності призводить до появи ознак деградації з боку сервера генерації навантаження. Це проявлялося у зростанні втрат ітерацій, збільшенні середнього часу відповіді та

погіршенні квантиля p95. На навантаженні 6700 RPS завантаження процесора на сервері генерації досягало 100 %, тоді як використання пам’яті залишалося в межах близько 4 GiB. Це свідчить про те, що обмежувальним чинником у верхній частині діапазону виступає саме генератор навантаження, а не сервери прикладного рівня. Результати тестування та ключові рішення для подальших випробувань наведені у табл. 4.1.

Таблиця 4.1 - Ключові орієнтири для верифікації стенда.

Показник	Значення
Базове навантаження для 10 000 слабо навантажених сайтів	1114 RPS
Штатний піковий режим (×2)	2228 RPS
Сценарій пік + HTTP flood 100 % від піку	4456 RPS
Контрольний режим, обраний для подальших експериментів	4800 RPS
Практична верхня межа стенда	Близько 6000 RPS
Запас відносно сценарію 4456 RPS до межі стенда	Близько 35 %
Запас контрольного режиму 4800 RPS відносно цільових 4456 RPS	7,7 %

Отже, навіть напружений сценарій, у якому до штатного пікового навантаження додається HTTP flood на рівні 100 % від піка, не виводить стенд на межу фізичних можливостей. Це дає змогу використовувати його для подальших досліджень без необхідності залучення потужнішого обладнання.

Після встановлення верхньої межі було виконано серію тривалих перевірок у контрольній точці 4800 RPS. Саме це значення обрано як робоче для верифікації стенда, оскільки воно перевищує розрахункову потребу подальших досліджень і водночас лишається нижче зони деградації сервера генерації навантаження. Для цього проведено три окремі одногодинні прогони для конфігурацій із 5, 10 та 15 екземплярами емуляторів сервера прикладного рівня (backend). Статистичні дані наведено у табл. 4.2, а показники навантаження зібрані у табл. 4.3.

Таблиця 4.2 - Результати одногодинних прогонів у контрольному режимі 4800 RPS.

Сервери	Навантаження на один інстанс	Втрати	Середній час відповіді	p95
5	≈960 RPS	0,00%	998,69 мс	1010,47 мс
10	≈480 RPS	0,00%	999,27 мс	1016,82 мс
15	≈320 RPS	0,00%	1000,61 мс	1024,20 мс

Таблиця 4.3 - Навантаження на сервери у контрольному режимі 4800 RPS.

Кількість інстансів	CPU генератора	CPU backend-сервера	RAM генератора	RAM backend-сервера
5	57-72%	40-45%	15%	40%
10	58-76%	42-46%	15%	40%
15	59-82%	43-46%	15%	40%

Отримані результати підтверджують, що у вибраному робочому режимі backend-рівень не є «вузьким місцем» стенду (рис 4.4). В усіх трьох одногодинних прогонах частка помилок становила 0.00%, а середній час відповіді залишався близьким до закладеної у модель величини 1 с. Це свідчить про те, що backend-емулятори мають достатній запас обчислювальних ресурсів для приймання запиту, формування сторінки обсягом 50 KiB та передавання відповіді без прихованого дефіциту процесорного часу чи оперативної пам’яті.

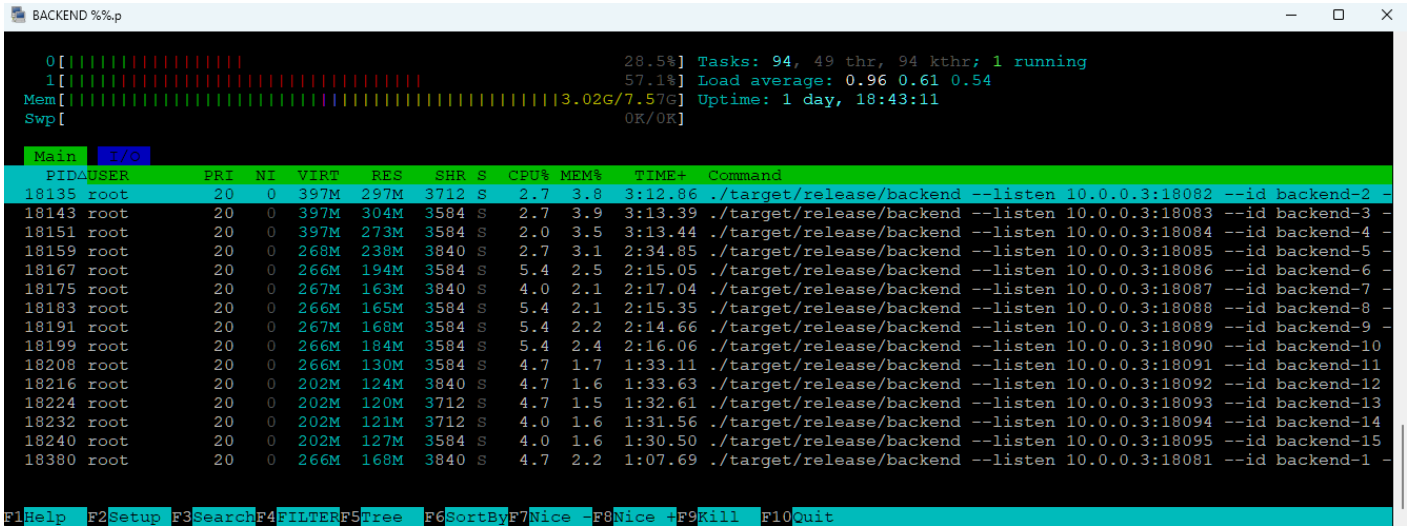

Рисунок 4.4 - Витрати ресурсу backend-емулятора під час навантаженні 4800 RPS.

Ще одним важливим результатом є оцінка пропускної здатності окремого віртуального backend-вузла. У конфігурації з п’яти екземплярів на один вузол припадало близько 960 RPS у повністю стабільному режимі, а під час пошуку верхньої межі стенду короткочасно досягався рівень понад 1200 RPS на один backend екземпляр (рис 4.5). Це істотно перевищує потрібну питому продуктивність для подальших експериментів з емуляцією хостингу слабо навантажених сайтів.

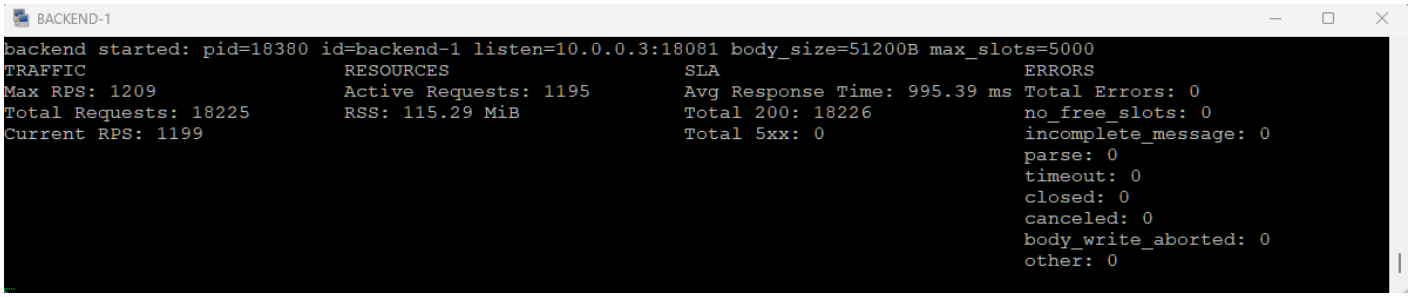

Рисунок 4.5 - Моніторингова консоль окремого backend-вузла у разі п’яти активних інстансів і навантаженні близько 6000 RPS.

Мережева інфраструктура також не виявилася обмежувальним фактором. За заданого розміру вихідної вебсторінки 50 КіБ та інтенсивності навантаження 4800 RPS (запитів за секунду) сумарна смуга пропускання для вихідного трафіку становить приблизно 1,9—2,0 Gbit/c, без урахування додаткових накладних витрат мережевих протоколів. За наявності каналу 10 Gbit/s це означає значний запас пропускної здатності. Отже, мережа між вузлами стенда не повинна впливати на результати подальших експериментів.

Таким чином, проведена верифікація підтвердила придатність експериментального стенда для наступних етапів дослідження. Сервер генерації навантаження забезпечує необхідну інтенсивність трафіку із запасом відносно цільового сценарію, backend-рівень не є «вузьким місцем» у робочому діапазоні навантажень, мережева інфраструктура має достатній резерв пропускної здатності. Це дає підстави вважати, що подальші результати визначатимуться вже властивостями розробленої інформаційної технології, а не побічними обмеженнями стенда.

4.4. Дослідження масштабованості балансувальника у багатодоменному TLS-середовищі

Одним із ключових чинників, що визначають придатність балансувальника до використання в кластерному вебхостингу, є його здатність працювати в умовах великої кількості доменів із захищеними з’єднаннями. У такому середовищі балансувальник повинен одночасно зберігати в оперативній пам’яті значний набір сертифікатів, виконувати вибір потрібного сертифіката за ім’ям домену, підтримувати стабільне

оброблення клієнтських запитів і зберігати працездатність під час зміни сертифікатного набору без зупинки процесу. Саме тому в цьому підрозділі досліджено два взаємопов’язані аспекти: вплив кількості доменів і сертифікатів на використання оперативної пам’яті та стійкість ILB під час масового оновлення сертифікатів під навантаженням.

Для побудови сертифікатного середовища використано спеціальний сценарій наведений у додатку 5. Цей сценарій формує повний ієрархічний ланцюжок сертифікації, що містить кореневий центр сертифікації, проміжний центр сертифікації та 10 000 окремих серверних сертифікатів для доменів вигляду site000001.example … site010000.example. Кореневий і проміжний сертифікати генеруються на основі еліптичної криптографії з кривою `secp384r1`, тоді як серверні сертифікати створюються на основі кривої `prime256v1`. Для кожного домену формується окрема пара ключів, унікальний серійний номер і розширення, що містить відповідне доменне ім’я. На боці сервера використовується файл `cert.pem`, який містить серверний і проміжний сертифікати у правильному порядку подання, тоді як кореневий сертифікат навмисно не додається до серверного пакету. Така побудова відтворює практику реального використання сертифікатного ланцюжка і дає змогу наблизити умови експерименту до багатодоменного середовища публічного хостингу.

Для усунення впливу зовнішніх чинників доменні імена тестового середовища розв’язувалися локально шляхом внесення записів до файлу `/etc/hosts`. Це допомогло запобігти затримкам та нестабільності зовнішнього DNS і забезпечити відтворюваність експериментів. Під час навантажувальних випробувань кожен клієнтський потік генератора навантаження був закріплений за конкретним доменом, що краще відповідає реальному режиму використання багатодоменного хостингу, ніж випадкова зміна доменного імені на кожному запиті.

4.4.1. Вплив кількості сертифікатів на використання оперативної пам’яті

Перший етап дослідження був присвячений оцінюванню впливу кількості завантажених сертифікатів на використання оперативної пам’яті процесом ILB. Вимірювання виконувались за показником RSS (Resident Set Size), який відображає обсяг сторінок процесу, що фактично перебувають у фізичній оперативній пам’яті [111]. Кількість доменів у стенді послідовно збільшувалась від десятків до 10 000, після чого для кожної конфігурації фіксувався обсяг пам’яті, зайнятий балансувальником.

Проведений експеримент показав монотонне зростання RSS зі збільшенням кількості завантажених сертифікатів. Характер цієї залежності в досліджуваному діапазоні виявився близьким до лінійного, що узгоджується з теоретичним очікуванням: кожний новий домен додає до стану балансувальника власний сертифікатний об’єкт, пов’язаний ключовий матеріал і службові структури, необхідні для вибору сертифіката та обслуговування захищеного з’єднання. Отриманий результат важливий з двох причин. По-перше, він підтверджує передбачуваність моделі зберігання сертифікатного стану в ILB. По-друге, він дає змогу заздалегідь оцінювати вимоги до оперативної пам’яті під час масштабування системи до великої кількості доменів.

У підсумку для цього етапу експерименту одержано залежність $RSS = f(N_{cert})$ (рис. 4.6), де N_{cert} — кількість активних сертифікатів.

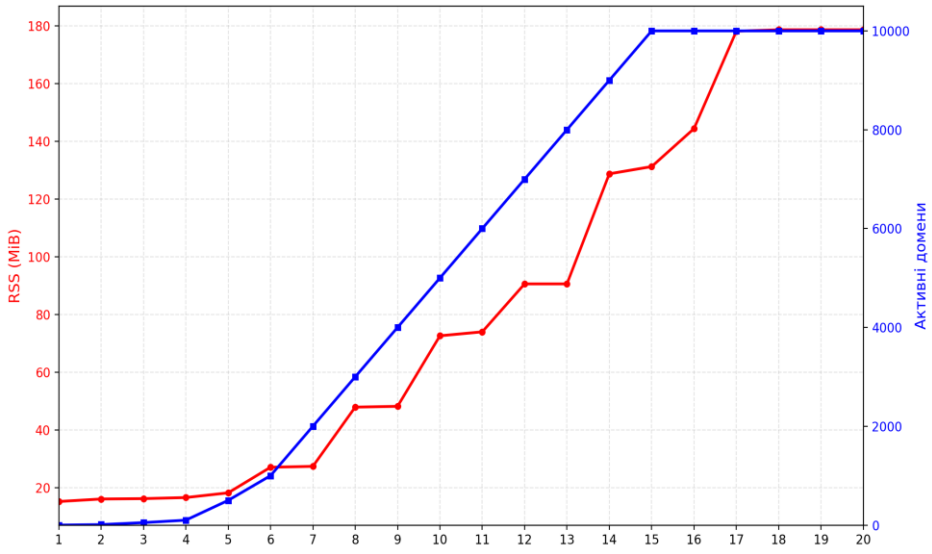

Рисунок 4.6 - Залежність витрат пам’яті від кількості доменів в обслуговуванні.

Важливо зазначити, що в цьому експерименті оцінювалися не витрати пам’яті після холодного старту балансувальника з наперед заданою кількістю сертифікатів, а витрати в режимі динамічного повного оновлення сертифікатного набору. Вимірювання виконувались без клієнтського навантаження, що дало можливість ізолювати внесок саме підсистеми оброблення сертифікатів у загальне використання оперативної пам’яті. Тому зафіксовані значення RSS враховують не лише зберігання активного набору сертифікатів, а й накладні витрати на зчитування та розбір сертифікатних файлів, завантаження ключового матеріалу, а також тимчасове співіснування старого і нового наборів під час їх атомарної заміни. За допомогою цього методу можна оцінити верхню межу витрат пам’яті для реалізованого механізму динамічного оновлення.

4.4.2. Стійкість ILB під час масового оновлення сертифікатів

Другий етап дослідження стосувався значно складнішого сценарію, а саме поведінки ILB під час масового оновлення сертифікатів під навантаженням без перезапуску процесу. У реальному середовищі вебхостингу зміна сертифікатного набору є звичайним явищем. Додаються нові домени, виконуються перевипуски сертифікатів, відбувається планова заміна ключового матеріалу. Для практичного використання балансувальника недостатньо лише один раз завантажити сертифікати на старті. Необхідно також гарантувати безпечне оновлення цього набору без втрати клієнтських запитів.

Початковий аналіз показав, що прямолінійне оброблення 10 000 сертифікатів створює суттєве короткочасне навантаження на процесор. За такого підходу система повинна зчитати з диска близько 20 000 файлів ключів і сертифікатів, розібрати їх, сформувати новий масив сертифікатного стану та атомарно підмінити ним попередній. У разі виконання всіх цих дій у вигляді щільного безперервного циклу утворюється ризик накопичення черги вхідних запитів, а в граничному режимі, деградації всього фронтального контуру через перевантаження процесора. Тому для ILB було реалізовано

інший підхід: оброблення сертифікатів не виконується одним піком, а розтягується в часі завдяки введенню коротких пауз між окремими операціями. Унаслідок цього повне оновлення 10 000 сертифікатів займає близько 10 секунд, але не створює руйнівного стрибка навантаження.

Слід окремо зазначити, що в підрозділі 4.3 було встановлено стійку роботу контуру формування навантаження, backend-рівня та мережевої інфраструктури на рівні 4800 RPS. Отже, це значення характеризує не граничні можливості балансувальника, а підтверджену пропускну здатність допоміжного контуру експериментального стенда. У сценарії, розглянутому в цьому підрозділі, подамо навантаження вже на балансувальник. Окрім звичайного оброблення клієнтських запитів, балансувальник одночасно підтримував багатодоменний режим захищених з'єднань і періодично виконував повне оновлення набору з 10 000 сертифікатів без зупинки процесу. Саме в цьому режимі було встановлено, що ILB є чутливішим елементом системи, ніж backend-рівень або контур формування навантаження. При збільшенні інтенсивності вхідного потоку понад 4400 RPS починали проявлятися ознаки перевантаження процесора та зниження продуктивності (рис. 4.7), що створювало ризик переходу до нестійкого режиму роботи. З цієї причини для подальших досліджень, пов'язаних із масовим оновленням сертифікатів під навантаженням, як межу тривалої стабільної роботи балансувальника було взято рівень 4400 RPS. Така постановка принципово важлива, оскільки дає змогу надалі інтерпретувати появу деградації саме як наслідок обмежень ILB, а не як результат недостатньої продуктивності генератора навантаження, backend-серверів чи мережі.

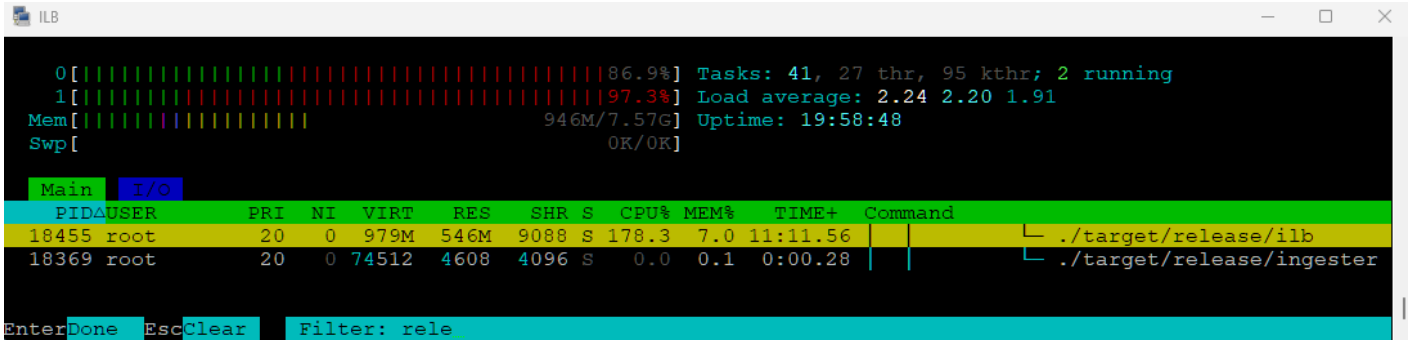

Рисунок 4.7 - Витрати ресурсів сервера ILB під час навантажені 4800 RPS

Експеримент виконувався у режимі реального трафіку на всі 10 000 доменів. Балансувальник працював під навантаженням 4400 RPS протягом однієї години, а процедура оновлення всього набору сертифікатів викликалась кожні 5 хвилин. Таким чином, протягом одного прогону система багаторазово проходила повний цикл перезавантаження всього сертифікатного середовища без зупинки процесу та без зменшення числа активних доменів. Важливо, що під час цього експерименту запити проходили повний робочий контур системи: маршрутизувалися на backend-рівень, а їх копії одночасно передавалися до асинхронного контуру аналізу, відбувалася перевірка IP-адреси у списку блокування і слухалась черга Redis. Отже, перевірка виконувалась не в спрощеному лабораторному режимі, а в умовах, наближених до штатної експлуатації. Результати цього експерименту наведено в табл. 4.4.

Таблиця 4.4 - Результати експерименту з динамічного оновлення сертифікатів.

Параметр	**Значення**
Кількість доменів	10 000
Інтенсивність вхідного потоку	4400 RPS
Тривалість прогону	1 година
Періодичність повного оновлення сертифікатів	кожні 5 хв
Обсяг одного оновлення	10 000 сертифікатів
Середній час відповіді	998,69 мс
Квантиль `p95`	999,56 мс
Частка помилок	0,00 %
Втрата клієнтських запитів	0
Завантаження процесора ILB	85–88 %

Як видно з наведених даних, навіть за умов регулярного повного оновлення сертифікатного набору балансувальник зберігав стійку працездатність. Втрата клієнтських запитів не зафіксована, а середній час відповіді та квантиль p95 залишалися на рівні базового режиму. Завантаження процесора перебувало в діапазоні 85–88 %, що відповідає рівню навантаження 4400 RPS і не демонструє додаткового критичного стрибка в моменти оновлення. Це свідчить про те, що запропонований механізм

розтягнутого в часі оновлення сертифікатів виявився достатнім для збереження працездатності ILB навіть у режимі інтенсивного багатодоменного трафіку.

Окремо було проаналізовано динаміку використання оперативної пам’яті одразу після запуску ILB з 10 000 сертифікатів, та під час послідовних оновлень. Отримані значення наведено в табл. 4.5.

Таблиця 4.5 - Результати експерименту з динамічного оновлення сертифікатів.

Стан системи	**RSS, MiB**
До першого оновлення	486,26
Після першого оновлення	519,56
Після другого оновлення	551,81
Після третього оновлення	552,16
Після наступних оновлень	Без істотного зростання

Початкове збільшення RSS після перших циклів оновлення пояснюється логікою безпечної заміни сертифікатного набору. Новий масив сертифікатів формується окремо від поточного робочого стану, після чого відбувається атомарна підміна посилання на нього. Такий підхід усуває ризик частково оновленого стану та забезпечує коректність вибору сертифіката в процесі обслуговування клієнтів. Водночас перші цикли оновлення вимагають додаткового резерву пам’яті для розміщення нової копії структур даних. Після того, як потрібний обсяг пам’яті було виділено й повторно використано, подальші оновлення вже не супроводжувалися помітним зростанням RSS. Це дає підстави трактувати зафіксоване збільшення не як витік пам’яті, а як наслідок первинного розширення робочих буферів у механізмі гарячого оновлення.

Отже, результати випробувань показали, що збільшення кількості сертифікатів у ILB супроводжується закономірним і передбачуваним зростанням використання оперативної пам’яті, тоді як масове оновлення 10 000 сертифікатів може виконуватись без втрати клієнтських запитів і без критичного погіршення часових характеристик. Це підтверджує, що розроблений балансувальник зберігає працездатність не лише у статичному багатодоменному режимі, а й у динамічному середовищі, де сертифікатний стан системи регулярно змінюється під час роботи.

4.5. Експериментальна перевірка модуля розрахунку ємності кластера та механізму масштабування

У цьому підрозділі виконано експериментальну перевірку центрального елемента запропонованої інформаційної технології, а саме програмної реалізації математичної моделі розрахунку ємності кластера та механізму адаптивного масштабування, обґрунтованих у розділі 2. На відміну від попередніх підрозділів, де оцінювалися переважно властивості окремих компонентів стенда або окремі експлуатаційні характеристики балансувальника, тут досліджується саме робота замкненого контуру керування. Його вхідними даними є поточна телеметрія вхідного потоку, а результатом — рішення про необхідну кількість активних backend-вузлів, що має забезпечити дотримання заданого рівня обслуговування за мінімально достатньої ємності кластера.

Метод перевірки побудовано так, щоб послідовно зіставити три рівні моделі з їх програмною реалізацією. На першому рівні перевіряється коректність згладжування телеметрії, оскільки саме згладжена оцінка інтенсивності вхідного потоку використовується як вхід для алгоритму планування ємності. На другому рівні виконується звіряння програмного розрахунку необхідної кількості слотів пам’яті та серверів із теоретичними значеннями, отриманими за рекурентною формулою Ерланга B. На третьому рівні досліджується вже динамічна поведінка системи: своєчасність додавання нових вузлів, коректність виведення надлишкових вузлів, відсутність автоколивань та вплив резервування ресурсів на втрати в перехідних режимах.

Параметри навантаження для цього блоку обрано з урахуванням результатів підрозділів 4.3 та 4.4. Тобто використано такий діапазон інтенсивностей, за якого експериментальний стенд уже підтвердив власну достатність, а балансувальник зберігає практичну працездатність без переходу в режим деградації через сторонні обмеження допоміжної інфраструктури. Це дає змогу інтерпретувати подальші результати саме як властивості реалізованого контуру керування ємністю, а не як наслідок браку продуктивності генератора навантаження, backend-рівня чи мережевого середовища.

4.5.1. Верифікація розрахунку ковзного вікна

Для перевірки коректності програмної реалізації механізму згладжування телеметрії було виконано порівняння значення `prediction_last_smoothed_rps`, що обчислюється в балансувальнику, із середнім значенням `current_rps` на попередньому ковзному вікні спостереження завдовжки 10 секунд. Після заповнення вікна отримано практичний збіг розрахованих значень, що підтверджує відповідність програмної реалізації формулі ковзного усереднення, наведеній у розділі 2. Невеликі залишкові відхилення пояснюються асинхронністю циклів збирання телеметрії, обчислення прогнозу та запису даних у журнал і не свідчать про помилку математичної моделі.

4.5.2. Верифікація програмної реалізації модуля прогнозування

Для перевірки коректності програмної реалізації модуля прогнозування було виконано звіряння розрахунку мінімально необхідної кількості слотів пам'яті M_{req} та кількості серверів K_{req} із теоретичними значеннями, отриманими за рекурентною формулою Ерланга B. Як контрольні точки було використано реальні значення `offered_traffic`, зафіксовані в телеметрії дослідного прототипу та зафіксовані у базі даних ClickHouse. Для кожної точки перевірялося, що для значення $M_{req} - 1$ ймовірність блокування перевищує цільовий поріг $\varepsilon = 0.001$, а для M_{req} вже виконується умова $E(A, M_r eq) <= \varepsilon$, що підтверджує мінімальність знайденої ємності (табл. 4.6).

Таблиця 4.6 - Порівняння теоретичного розрахунку кількості слотів та практичної реалізації.

Контрольна точка	A **з БД**	M_{req} **з БД**	M_{req} **теор.**	**E(A,M-1)**	**E(A,M)**
1	400.2	450	450	0.001085	0.000964
2	430.0	481	481	0.001116	0.000997
3	450.2	503	503	0.001015	0.000908
4	500.0	555	555	0.001022	0.000920
5	575.0	633	633	0.001038	0.000942

Подальший перехід до кількості серверів за формулою $K_{req} = \lceil M_{req}/C_{node} \rceil$ також показав повний збіг із даними, зафіксованими в базі телеметрії, що підтверджує коректність реалізації алгоритму розрахунку ємності кластера (табл. 4.7). Для пришвидшення цієї перевірки в тестовому сценарії час очікування готовності нового сервера було штучно зменшено до 1 с, що відображається у значеннях $K_{pending}$.

Таблиця 4.7 - Порівняння теоретичного розрахунку кількості серверів та практичної реалізації.

Контрольна точка	**K_{req} з БД**	**K_{req} теор.**	**K_{curr}**	**$K_{pending}**	**ΔK**
1	2	2	2	0	0
2	3	3	2	1	1
3	3	3	2	1	0
4	3	3	3	0	0
5	3	3	3	0	0

4.5.3. Верифікація моделі масштабування

На відміну від п. 4.5.2, у якому було підтверджено коректність самого розрахунку необхідної ємності кластера, у цьому підрозділі перевіряється динамічна поведінка системи керування, тобто своєчасність додавання серверів та коректність їх виведення. Для цього було виконано навантажувальний експеримент, у якому сумарний трафік плавно зростав від 400 до 1400 запитів за секунду, після чого так само плавно зменшувався до початкового рівня. Загальна тривалість прогону становила одну годину. Резервування потужності на цьому етапі було встановлено на рівні 0 %, щоб оцінити роботу алгоритму саме в його базовій конфігурації, без додаткового штучного запасу.

На рисунку 4.8 червоною лінією показано динаміку сумарного трафіку, а синьою фактичну кількість активних backend-серверів. Із графіка видно, що в процесі зростання навантаження кластер послідовно збільшується з 2 до 7 серверів, а під час спаду трафіку кількість активних вузлів у зворотному порядку зменшується до 2. Сам по собі такий результат уже підтверджує коректність виконавчої логіки масштабування, однак для правильної інтерпретації важливо брати до уваги, що на графіку відображено саме

активні сервери, тобто ті, що вже завершили етап під'єднання і почали приймати трафік. Рішення про додавання нового вузла приймається раніше, а момент появи нового ступеня на графіку відповідає вже завершенню його ініціалізації. Отже, візуальне зміщення між моментом збільшення навантаження і моментом зростання кількості активних серверів є очікуваним і пояснюється часовими параметрами моделі, а не помилкою алгоритму.

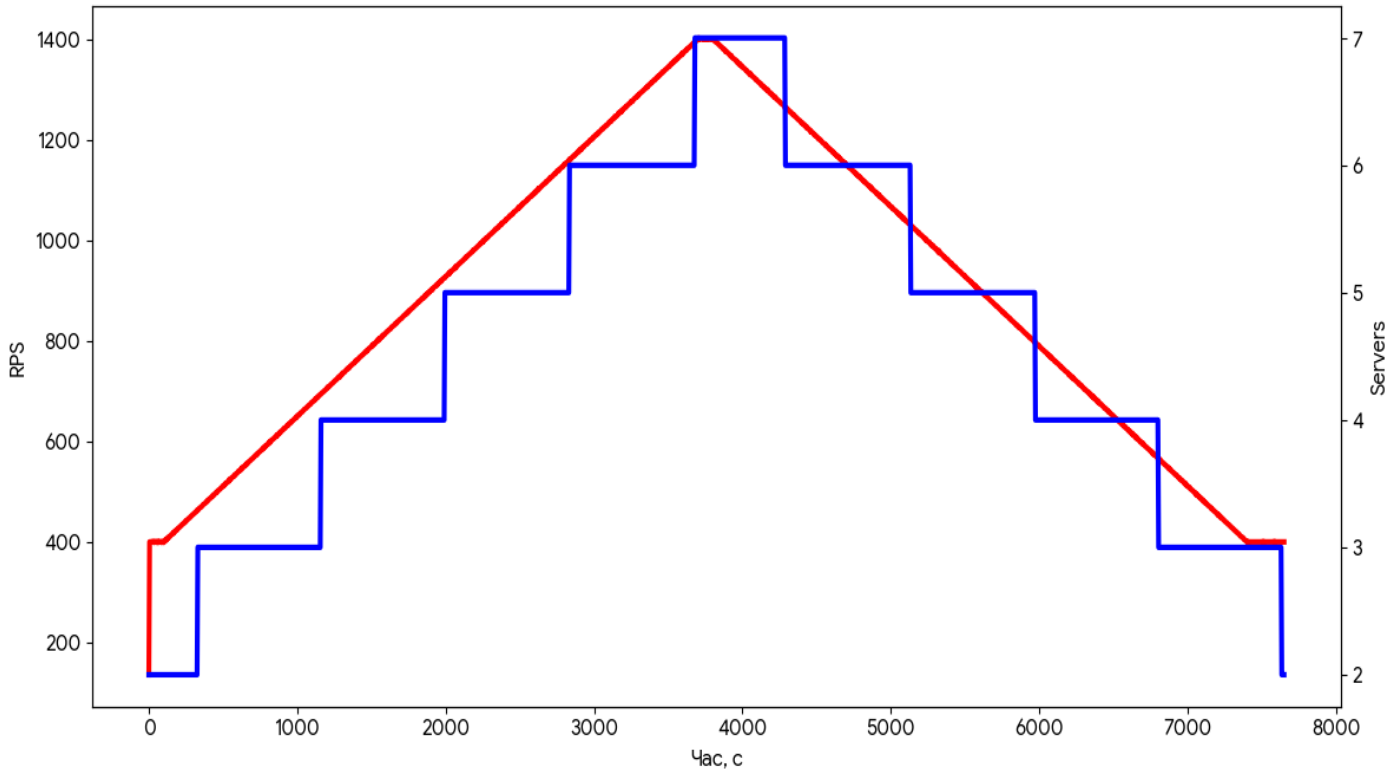

Рисунок 4.8 - Динаміка зміни загального трафіку та масштабу кластера.

Характер зміни масштабу кластера добре узгоджується з логікою, закладеною в розділі 2. Під час зростання навантаження нові вузли додаються не хаотично, а в ті моменти, коли поточна конфігурація наближається до межі допустимого режиму. Після введення нового сервера сумарне навантаження перерозподіляється між більшою кількістю вузлів, унаслідок чого навантаження на кожен окремий сервер миттєво зменшується. Відтак, у міру продовження зростання вхідного потоку, це навантаження знову плавно підвищується до наступного порогового рівня. Саме тому на рис. 4.9

навантаження окремих серверів має характерну пилкоподібну форму, яка є прямим наслідком послідовного масштабування вгору.

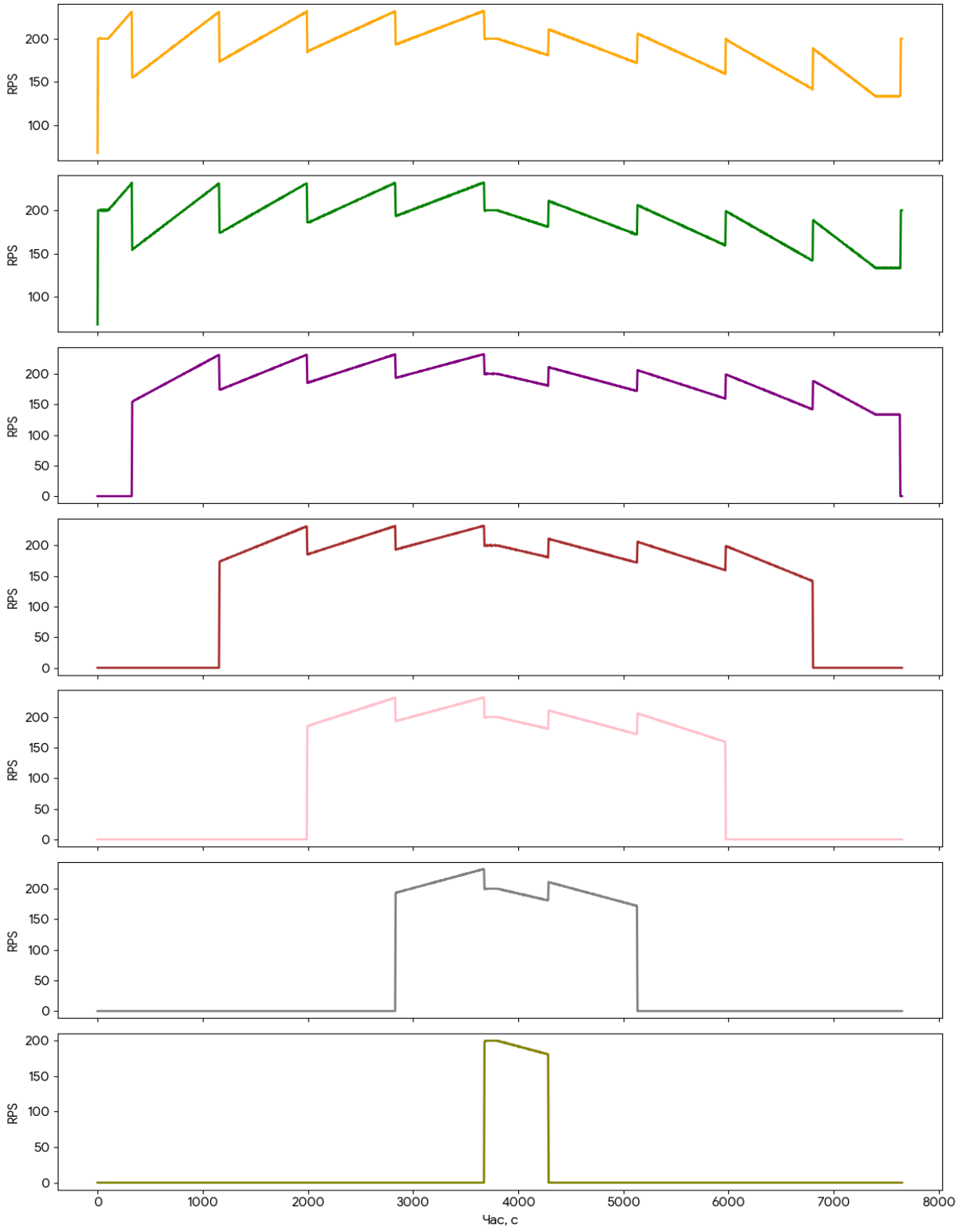

Рисунок 4.9 - Динаміка навантаження на окремі сервери кластера.

Другий важливий результат полягає в тому, що в разу зменшення сумарного трафіку масштабування вниз відбувається не симетрично до масштабування вгору. Вузли не вимикаються одразу після першого ж зменшення навантаження, а виводяться з кластера із запізненням. Така поведінка правильна, оскільки відповідає принципу гістерезису і запобігає автоколиванням. На рис. 4.8 це проявляється тим, що після досягнення піка навантаження кластер певний час зберігає надлишкову, з погляду миттєвого RPS, кількість серверів, і лише після стійкого зниження трафіку переходить до зменшення масштабу.

Рис. 4.9 дає можливість детальніше проаналізувати поведінку окремих вузлів. Перші сервери працюють протягом усього експерименту, тоді як наступні послідовно вводяться в експлуатацію у міру зростання навантаження. Після активації кожний новий сервер одразу отримує свою частку трафіку, а навантаження на вже активні вузли відповідно зменшується. Під час масштабування вниз спостерігається зворотний процес: навантаження на окремий сервер спочатку знижується до нуля, після чого сервер виводиться з пулу. Це свідчить про коректну реалізацію режиму `draining`, коли вузол не вимикається миттєво, а спочатку припиняє отримувати нові запити і завершує обслуговування вже прийнятого навантаження. Така поведінка принципово важлива для практичного використання системи, оскільки допомагає уникати штучних втрат запитів під час зменшення масштабу кластера. Сервер генерації навантаження на базі k6 також підтвердив відсутність втрати запитів.

В підрозділі 4.3 уже було підтверджено достатню продуктивність сервера генерації навантаження, backend-рівня та мережевої інфраструктури для режимів, близьких до досліджуваного діапазону. Тому виявлена на графіках динаміка визначається саме роботою балансувальника та його контуром керування, а не обмеженнями допоміжної інфраструктури експерементального стенда. Це надає результатам підрозділу методичної чистоти й дає можливість безпосередньо пов’язувати спостережувану поведінку кластера з реалізацією модулів `prediction` і `scaling`.

Отримані результати підтверджують кілька принципових властивостей розробленої моделі масштабування. По-перше, механізм масштабування вгору забезпечує своєчасне нарощування кількості серверів у відповідь на стійке зростання навантаження. По-друге, механізм масштабування вниз працює з контрольованим запізненням, що усуває небажані коливання складу кластера. По-третє, фактичний розподіл трафіку між вузлами після кожної зміни масштабу залишається впорядкованим і передбачуваним. У сукупності це свідчить про те, що програмна реалізація модуля масштабування коректно відтворює логіку математичної моделі та забезпечує стійке регулювання ємності кластера в умовах змінного навантаження.

4.5.4. Верифікація впливу резервування ресурсів

Після перевірки коректності згладжування телеметрії, модуля прогнозування та виконавчої логіки масштабування було проведено окреме дослідження впливу резервування ресурсів на стійкість кластера в перехідних режимах. Мета цього етапу — з’ясувати, якою мірою додатковий резерв обчислювальної ємності зменшує втрати запитів у ситуації, коли вхідний потік зростає швидше, ніж кластер встигає під’єднати нові сервери.

У поточній конфігурації прототипу довжина вікна згладжування становила 10 с, перерахунок прогнозу виконувався раз на 1 с, а час ініціалізації нового backend-сервера брався рівним 90 с. Отже, для консервативного оцінювання часової реакції системи використано величину

$$\tau_{react} \approx 10 + 1 + 90 = 101 \text{ с.}$$

Кожний backend-сервер у дослідному кластері мав апаратне обмеження 240 конкурентних слотів. Згідно з моделлю Erlang B, за умови цільового рівня обслуговування $SLA = 99{,}9\%$ практично допустиме навантаження на один сервер становить близько $202{,}6$ запитів за секунду. Це значення є меншим за фізичну межу 240 слотів, оскільки для випадкового потоку запитів робота поблизу повного насичення ресурсу вже супроводжується неприйнятно високою ймовірністю втрат.

Слід зазначити, що в цьому підрозділі не вводиться одна універсальна безпечна швидкість зростання трафіку. Причина полягає в тому, що момент появи втрат визначається не лише часовою затримкою реакції контуру керування, а й поточним рівнем навантаження, числом уже активних серверів, моментом спрацювання порога масштабування та кількістю вузлів, які замовляються одночасно. Тому стійкість системи до реактивного зростання навантаження далі оцінюється не суто аналітично, а на основі серії натурних експериментів.

Як форсований випробувальний профіль було обрано лінійне зростання трафіку зі швидкістю 2 RPS за секунду від нуля до 4450 RPS, що відповідає приблизно 120 RPS за хвилину. Такий профіль досягає верхньої межі приблизно за 37 хвилин. Для порівняння, для сукупності з 10 000 слабо навантажених сайтів із середнім навантаженням 0,114 RPS на сайт базовий рівень становить близько 1140 RPS. Отже, за такого темпу росту сумарне навантаження подвоюється приблизно за 9,5 хвилини, що є суттєво агресивнішим сценарієм, ніж типове природне зростання легітимного трафіку. Саме тому він був використаний як стресовий профіль для оцінювання запасу стійкості кластера.

Резервування в цьому дослідженні визначалося не як частка поточної ємності всього кластера, а як додаткова кількість слотів, еквівалентна певній частці місткості одного сервера. Якщо $C_{node} = 240$ слотів, то при резерві $p\%$ до розрахункової потреби додається:

$$M_{res} = C_{node} \cdot \frac{p}{100}.$$

Тоді планова місткість визначається як

$$M_{plan} = M_{req} + M_{res},$$

а кількість серверів для такого режиму обчислюється за формулою

$$K_{plan} = \left\lceil \frac{M_{plan}}{C_{node}} \right\rceil.$$

Наприклад, резерв 10 % відповідає додатковим 24 слотам, а резерв 20 % — 48 слотам. Для кожного значення резерву було виконано по три прогони, після чого в таблицях наведено середні значення втрат.

Результати першої серії експериментів, у якій змінювався лише розмір резерву за фіксованої швидкості зростання 2 RPS/c, подано в табл. 4.8.

Таблиця 4.8 - Залежність втрат від розміру резервування під час швидкості зростання трафіку 2 RPS/c

Резерв	Швидкість росту	Втрати	Досягнутий SLA
0%	2,0 RPS/c	2,72%	97,28%
10%	2,0 RPS/c	1,80%	98,20%
20%	2,0 RPS/c	1,03%	98,97%
25%	2,0 RPS/c	0,76%	99,24%
30%	2,0 RPS/c	0,52%	99,48%
35%	2,0 RPS/c	0,25%	99,75%
40%	2,0 RPS/c	0,01%	99,99%

Наведені дані показують чітку закономірність: зі збільшенням резерву втрати в перехідному режимі монотонно зменшуються. За відсутності резерву кластер у форсованому режимі збільшення трафіку втрачає в середньому 2,72 % запитів за весь період росту. Введення навіть 10 % резерву помітно покращує результат, однак цього ще недостатньо для наближення до цільового рівня 99,9 %. При резерві 20 % втрати зменшуються приблизно в 2,6 раза порівняно з режимом без резерву, але й цього замало для досягнення цільового показника. Лише при резерві 40 % для вибраного стресового профілю фактично досягається режим, близький до безвтратного, із середнім значенням $SLA = 99{,}99\%$.

Отже, резервування справді працює як механізм зменшення втрат у перехідних режимах, але його ефективність суттєво залежить від швидкості зростання трафіку. Сам по собі резерв 10—20 % не гарантує досягнення цільового рівня 99,9 % у форсованому сценарії 2 RPS/c, хоча і різко зменшує глибину деградації.

У другій серії експериментів розмір резерву було зафіксовано на рівні 20 %, а змінювалася вже швидкість зростання трафіку. Метою цього етапу було оцінити, наскільки чутливий кластер до темпу зміни навантаження навіть за наявності помірного резерву. Результати табл. 4.9 показують, що за фіксованого резерву 20 % зменшення швидкості росту трафіку істотно покращує якість обслуговування. Перехід від 2,0 до 1,5

RPS/c зменшує втрати більш ніж у шість разів, а при 1,0 RPS/c у проведених прогонах втрат не зафіксовано. Це означає, що перехідні втрати визначаються не лише розміром резерву, а й співвідношенням між темпом збільшення навантаження та часовою реакцією механізму масштабування.

Таблиця 4.9 - Залежність втрат від швидкості зростання трафіку під час резервуванні 20 %

Резерв	Швидкість росту	Втрати	Досягнутий SLA
20%	2,0 RPS/c	1,03%	98,97%
20%	1,5 RPS/c	0,16%	99,84%
20%	1,0 RPS/c	0,00%	100,00%

Таким чином, у межах натурного експерименту підтверджено практичний наслідок положень підрозділу 2.2 щодо інерційності контуру керування: за фіксованого значення τ_{react} втрати зростають зі збільшенням швидкості приросту трафіку і, навпаки, зменшуються зі збільшенням резерву, який компенсує часову затримку введення нових вузлів у роботу.

У межах проведених експериментів також спостерігалась ще одна важлива тенденція: за вищого поточного навантаження кластер виявлявся стійкішим до подальшого реактивного зростання трафіку. Це пояснюється тим, що в таких режимах у системі вже активна більша кількість серверів, а модуль масштабування може одночасно ініціювати підключення кількох нових вузлів. Унаслідок цього відносний вплив затримки введення одного окремого сервера зменшується, а кластер краще витримує короткочасне прискорене зростання вхідного потоку.

Практичний висновок з отриманих результатів полягає в тому, що резервування ресурсів у кластері доцільне і необхідне саме як засіб пом’якшення перехідних режимів. Водночас розмір цього резерву слід вибирати зважаючи на очікувану динаміку навантаження. Для досліджуваного прототипу резерв, еквівалентний 20 % місткості одного сервера, уже істотно знижує втрати, однак для дуже агресивного профілю зростання 2 RPS/c цього недостатньо для досягнення $SLA = 99{,}9\%$. Для такого сценарію практично безвтратний режим забезпечувався збільшенням резерву до 40 %.

На відміну від класичної моделі хостингу, у якій запас потужності зазвичай доводиться підтримувати на кожному окремому сервері, у кластерній архітектурі резерв задається централізовано, як надбавка до розрахункової ємності всього пулу. Саме це є однією з переваг кластерного підходу, оскільки дає змогу зменшити сумарний обсяг ресурсу, який простоює, й водночас зберігати прийнятну стійкість у перехідних режимах.

Отже, натурні експерименти підтвердили, що резервування ресурсів істотно впливає на стійкість кластера за швидкого зростання трафіку. Збільшення резерву послідовно зменшує втрати, а зменшення темпу росту навантаження додатково покращує якість обслуговування. Це підтверджує доцільність внесення параметра резервування до практичної конфігурації модуля розрахунку ємності та показує, що його вибір має здійснюватися зважаючи не лише на стаціонарне навантаження, а й динаміку перехідних процесів.

Загалом результати підрозділу 4.5 підтверджують узгодженість програмної реалізації з математичною моделлю, наведеною в розділі 2. Експериментально встановлено коректність згладжування телеметрії, правильність розрахунку мінімально необхідної ємності кластера за формулою Ерланга B, адекватність виконавчої логіки масштабування в режимах `scale-out` і `scale-in`, а також істотний вплив резервування на втрати в перехідних режимах. Це дає підстави стверджувати, що модулі `prediction` і `scaling` не лише коректно відтворюють окремі формули моделі, а й забезпечують стійке регулювання ємності кластера в умовах змінного та реактивно зростаючого навантаження.

4.6. Порівняльне дослідження ресурсної ефективності класичного режиму спільного хостингу та адаптивного кластерного підходу

У цьому підрозділі виконано порівняння двох архітектур обслуговування масиву слабо навантажених вебресурсів. Перша архітектура відповідає класичному режиму спільного хостингу, у якому кожен сервер обслуговує статично закріплену за ним групу сайтів. Друга архітектура ґрунтується на спільному пулі backend-вузлів із динамічним

масштабуванням, де кількість активних серверів змінюється відповідно до поточного навантаження. Метою дослідження — оцінити, чи дає кластерний підхід виграш за інтегральними витратами оперативної пам’яті за збереження майже однакового рівня обслуговування.

4.6.1. Формування тижневих профілів навантаження

Для проведення експерименту використано сценарний підхід до формування навантаження. Його сутність полягає не у відтворенні статистики конкретного провайдера, а у побудові керованого й відтворюваного тижневого часового ряду, який відображає характерні добові та тижневі зміни активності користувачів [112,113]. Такий підхід доцільний для дисертаційного дослідження, оскільки дає змогу поєднати реалістичність форми навантаження з повним контролем над його параметрами.

На першому етапі було сформовано шість базових тижневих профілів для різних типів вебсайтів. Кожен профіль задавав лише форму зміни навантаження в часі, а не абсолютний рівень трафіку. Тому всі шаблони нормувалися в діапазоні від 0 до 100 умовних одиниць. Форма профілів задавалася відповідно до очікуваної поведінки аудиторії: денна активність для офісних ресурсів, вечірні підсилення для контентних сайтів, зменшення активності у вихідні для частини типів ресурсів, а також короткі локальні піки для новинних сайтів. Отримані шаблони наведено на рис. 4.10.

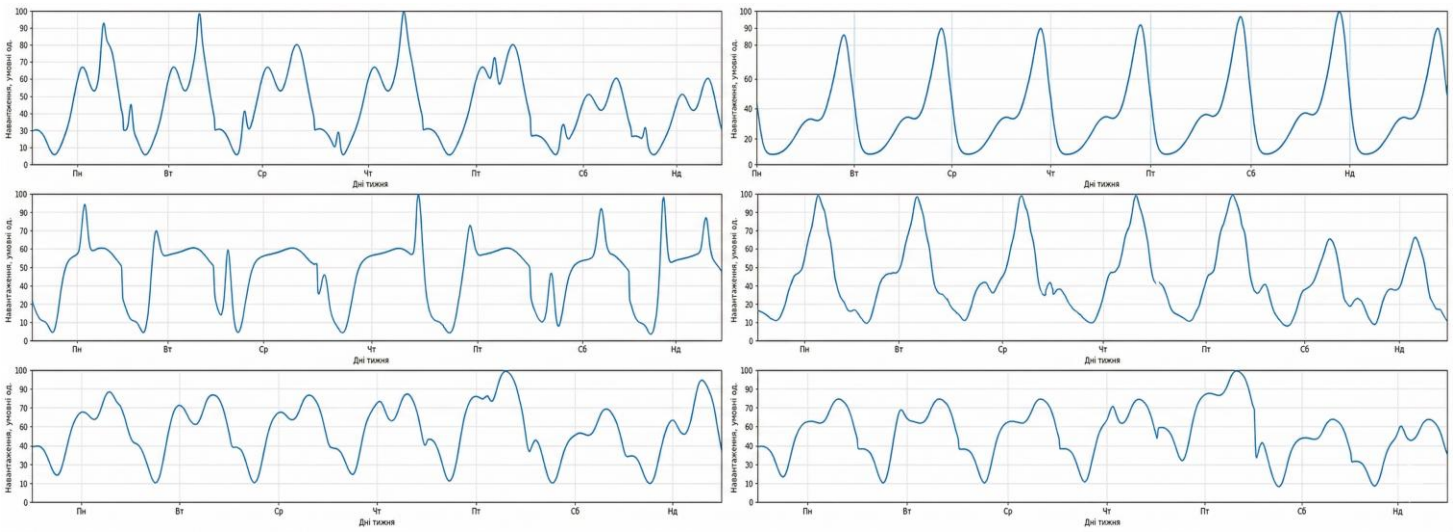

Рисунок 4.10 - Тижневий профіль навантаження для різних типів сайтів.

На другому етапі кожен базовий шаблон було перетворено на п’ять окремих профілів сайтів. Для цього до значення кожної часової точки додавалося випадкове локальне відхилення в межах ±10 %. Після цього до профілю застосовувався циклічний часовий зсув на −2, −1, 0, +1 та +2 години. Такий зсув дав змогу моделювати групи користувачів із різними часовими звичками та уникнути штучного синхронного збігу піків. Додатково для кожного сайту задавався власний коефіцієнт масштабу в діапазоні від 0,2 до 5, що відображало різницю в популярності й розмірах аудиторії навіть усередині одного типу ресурсів.

На третьому етапі п’ять профілів, отриманих із одного шаблону, підсумовувалися в один сумарний профіль відповідного типу сайтів. Після цього всі сумарні профілі типів підсумовувалися в єдиний глобальний профіль навантаження та нормалізовано. Саме цей профіль було збережено у файлі `profile_global_normalized_0_100.csv` (додаток 7) і надалі використано як фіксований вхідний сценарій для всіх експериментів. Отже, випадковість була присутня лише на етапі генерації шаблону, а всі натурні випробування виконувалися вже на зафіксованому часовому ряді, що забезпечило повну відтворюваність результатів. Візуалізацію сумарного профілю наведено на рис. 4.11.

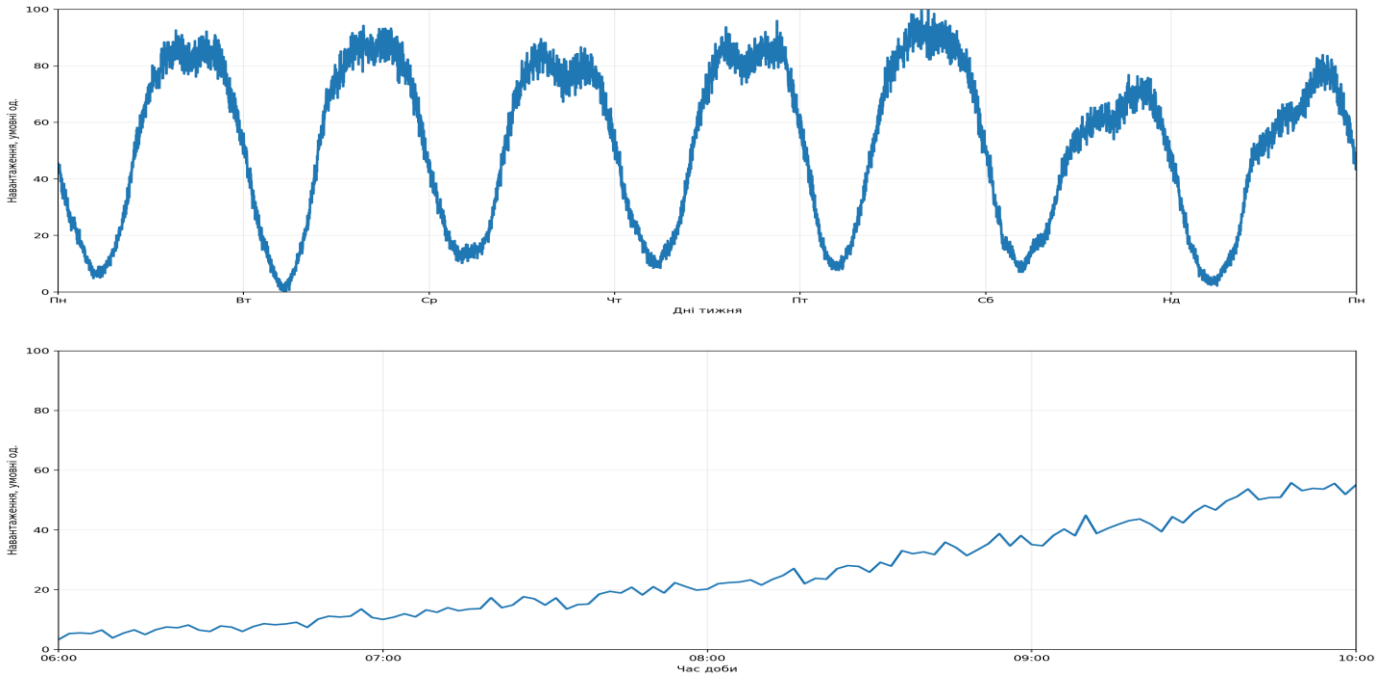

Рисунок 4.11 - Сумарний тижневий профіль навантаження.

Дискретизацію профілю було обрано рівною 2 хвилинам. Це дало 5040 часових інтервалів на один тиждень і допомогло сформувати довгий, але технічно відтворюваний сценарій для генератора навантаження. Такий крок достатньо точно передає повільні добові й тижневі коливання, не створюючи надмірного навантаження на систему генерації трафіку.

Для переходу від умовних одиниць до реальних значень інтенсивності було прийнято, що пікове навантаження одного сайту дорівнює подвоєному середньому галузевому значенню `0,114 RPS`[30], тобто

$$r_{site}^{max} = 2 \cdot 0{,}114 = 0{,}228 \text{ RPS}.$$

Тоді для довільної кількості сайтів N реальний профіль навантаження визначався за формулою

$$\Lambda_N(t) = \frac{P(t)}{100} \cdot N \cdot r_{site}^{max},$$

де $P(t)$ — значення нормованого сумарного профілю в момент часу t.

За даними файлу `profile_global_normalized_0_100.csv` сумарний профіль має мінімум `15,649`, середнє значення `53,072` і максимум `100,000`. Тому середнє навантаження одного сайту в експериментальному сценарії становить

$$\bar{r}_{site} = 0{,}228 \cdot 0{,}53072 \approx 0{,}121 \text{ RPS}.$$

Отже, сформований експериментальний профіль виявився приблизно на 6,1 % інтенсивнішим за базове галузеве значення `0,114 RPS` [30]. Це означає, що дослідження проводиться не за заниженим, а за помірно консервативним сценарієм.

4.6.2. Калібрування місткості одного сервера та побудова порівнюваних сценаріїв

Після формування нормованого тижневого профілю було виконано його масштабування для двох типів експериментів. Перший тип використовувався для визначення місткості одного сервера в режимі спільного хостингу. Другий тип застосовувався вже для безпосереднього порівняння статичної багатосерверної схеми з адаптивним кластером.

Параметри одного сервера в обох архітектурах було взято однаковими. Сервер має 64 GiB оперативної пам’яті, з яких 4 GiB резервуються під операційну систему, а 60 GiB доступні для прикладного навантаження. За прийнятого в дисертації припущення про 256 MiB пам’яті на один одночасний запит це відповідає 240 конкурентним слотам на один сервер. Отже, порівняння статичного й кластерного режимів проводилося для еквівалентних вузлів прикладного рівня.

Для калібрування місткості одного сервера було використано базовий профіль навантаження, масштабований під різну кількість умовних сайтів. Знайдений фінальний масштаб відповідає такій кількості сайтів, за якої один сервер ще забезпечує цільовий тижневий рівень обслуговування 99,9 %, але додавання ще одного сайту вже приводить до порушення цієї умови. Унаслідок натурного експерименту було встановлено, що для сформованого тижневого профілю граничне значення становить 1082 сайти на один сервер.

Принципово важливо, що число 1082 не виводилося прямим діленням кількості слотів сервера на навантаження одного сайту. Такий підхід був би надто грубим, оскільки реальний тижневий профіль містить часові зсуви, локальні випадкові варіації, неповний збіг піків окремих сайтів і виражену добову та тижневу циклічність. Тому місткість одного сервера визначалася саме натурним експериментом на зафіксованому тижневому профілі. За результатами цього калібрування встановлено, що при 1082 сайтах сервер ще підтримує цільовий тижневий SLA, тоді як додавання ще одного сайта, тобто збільшення пікового рівня на 0,228 RPS, уже переводить систему нижче межі 99,9 %. Отже, значення 1082 слід трактувати як експериментально відкалібровану місткість одного сервера для сформованого профілю навантаження.

Після цього для порівняльного експерименту було сформовано сукупність із

$$N_{total} = 9 \cdot 1082 = 9738$$

сайтів. У статичній схемі спільного хостингу такий обсяг навантаження відповідає дев’яти окремим серверам, а в кластерній схемі ті самі 9 738 сайтів обслуговуються

спільним пулом backend-вузлів із динамічним масштабуванням. Для цього сценарію було побудовано профіль, для якого пікове навантаження дорівнює

$$9738 \cdot 0{,}228 = 2220{,}264 \text{ RPS}.$$

Отже, обидві архітектури порівнювалися на одному й тому самому тижневому профілі, але в однаковому масштабі 9738 сайтів, що забезпечує коректність подальшого зіставлення.

Основні характеристики профілів, використаних у подальших експериментах, наведено в табл. 4.10.

Таблиця 4.10 - Основні характеристики тижневих профілів навантаження.

Профіль	Сайтів	Мінімум	Середнє	Максимум	Призначення
Нормований	-	15,649	53,072	100,000	Базовий шаблон
Один сервер	1 082	38,604 RPS	130,926 RPS	246,696 RPS	Тестування одного сервера
Кластер	9 738	347,440 RPS	1178,337 RPS	2220,264 RPS	Тестування кластера

4.6.3. Порівняння статичної та кластерної архітектур

Для моделі спільного хостингу було виконано тижневий натурний експеримент для одного сервера з 1082 сайтами. Оскільки порівняння з кластером проводилося для 9738 сайтів, отримані результати було масштабовано на дев’ять однакових серверів. У цій схемі всі сервери є постійно активними протягом усього тижня, тому сумарні витрати оперативної пам’яті для backend-рівня залишаються сталими. За 9 серверів по 64 GiB маємо постійну ємність 576 GiB, а інтегральний показник використання пам’яті за 5040 інтервалів по 2 хвилини становить

$$F_{shared} = 576 \cdot 5040 = 2\,903\,040 \text{ GiB·інтервалів}.$$

Для кластерної архітектури використовувався той самий набір із 9738 сайтів і той самий тижневий профіль, але з увімкненими модулями прогнозування та масштабування. Резервування ресурсів на цьому етапі було встановлено на рівні 0 %, щоб умови порівняння залишалися еквівалентними до статичної схеми, де додатковий запас також не закладався. На рисунку 4.12 наведено результат тестування кластерної архітектури,

де видно співвідношення кількості серверів до рівня навантаження, а також моменти (червоні вертикальні лінії) коли кластер втрачав запити через нестачу ресурсів.

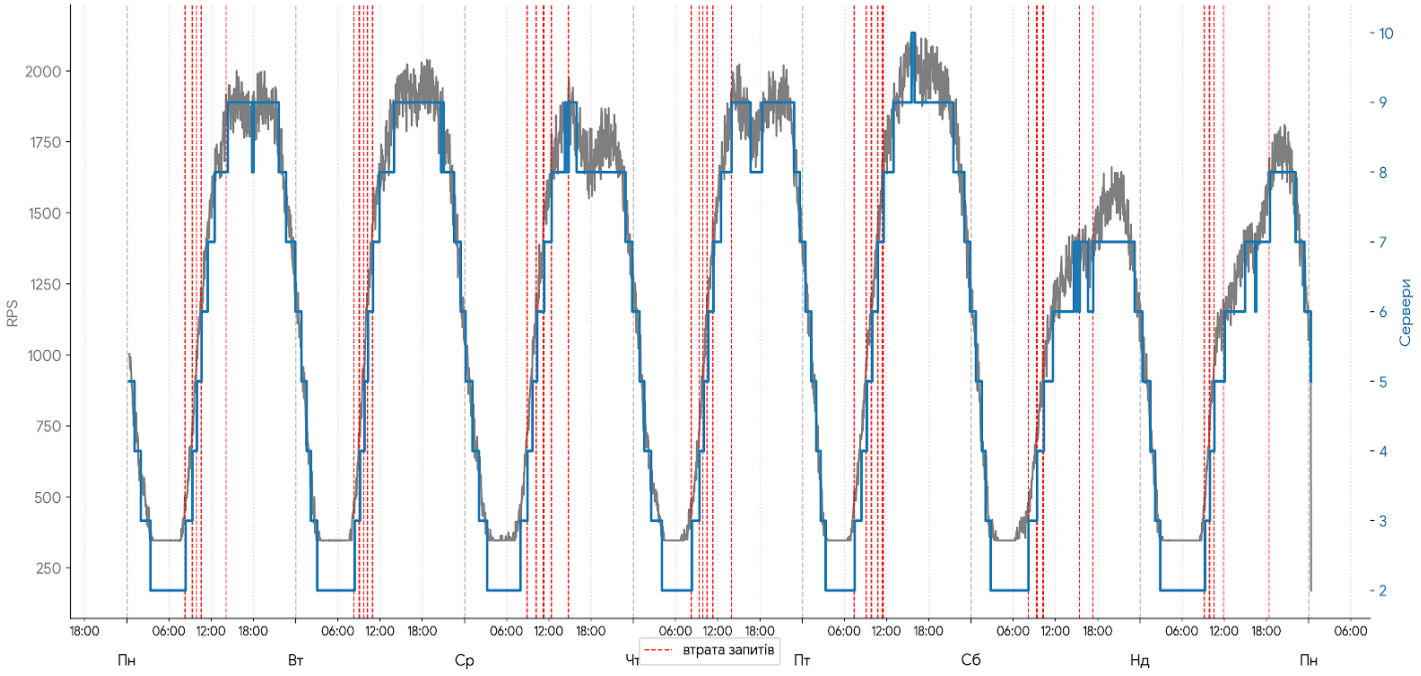

Рисунок 4.12 - Результат тестування кластерної архітектури.

За результатами експерименту інтегральні витрати пам’яті backend-рівня становили:

$$F_{cluster,backend} = 1\,846\,784 \text{ GiB·інтервалів.}$$

Оскільки практична схема передбачає дублювання балансувальника, до інтегральних витрат було додано два екземпляри ILB по 8 GiB кожен, тобто

$$F_{ILB} = 16 \cdot 5040 = 80\,640 \text{ GiB·інтервалів.}$$

Тоді сумарні інтегральні витрати пам’яті для кластерної архітектури дорівнюють

$$F_{cluster,total} = 1\,846\,784 + 80\,640 = 1\,927\,424 \text{ GiB·інтервалів.}$$

Для зручності сприйняття результати порівняння наведено в табл. 4.11.

Таблиця 4.11 - Порівняння статичної схеми спільного хостингу та адаптивного кластера

Показник	Спільний хостинг	Кластерний підхід
Загальна кількість сайтів	9738	9738
Кількість backend-серверів	9 постійно активних	динамічно змінюється
Надіслано запитів за тиждень	712 540 800	712 540 920

Показник	Спільний хостинг	Кластерний підхід
Втрачених запитів	165 924	176 880
Успішно оброблено запитів	712 374 876	712 364 040
Тижневий SLA, %	99,9767	99,9752
Інтегральні витрати RAM, GiB·інтервалів	2 903 040	1 927 424

З наведених даних видно, що обидві архітектури забезпечили практично однаковий рівень якості обслуговування. Для статичної схеми спільного хостингу тижневий SLA становив 99,9767 %, а для кластерної архітектури 99,9752 %. Відмінність між ними дуже мала і не змінює головного висновку: за однакового цільового рівня обслуговування кластерна схема потребує помітно менших інтегральних витрат оперативної пам’яті.

Інтегральний коефіцієнт ресурсної ефективності кластерної архітектури відносно статичної схеми обчислювався за формулою

$$\xi_{consolidation} = 1 - \frac{F_{cluster,total}}{F_{shared}}.$$

Підставляючи отримані значення, маємо

$$\xi_{consolidation} = 1 - \frac{1\,927\,424}{2\,903\,040} \approx 0{,}336,$$

тобто економія інтегральних витрат оперативної пам’яті становить близько 33,6 %.

Зміст цього результату можна інтерпретувати і в більш наочній формі. Статична схема протягом усього тижня утримує дев’ять серверів, незалежно від поточного навантаження. Натомість кластерна схема фактично використовує в середньому еквівалент приблизно 5,73 backend-сервера, або близько 5,98 серверних еквівалентів по 64 GiB, якщо врахувати два балансувальники. Саме ця динамічна зміна кількості активних вузлів і забезпечує виграш за інтегральним використанням оперативної пам’яті. Отриманий виграш є прямим наслідком того, що модуль масштабування, описаний у підрозділі 3.3, у кожен момент часу підтримує не довільну, а мінімально достатню ємність кластера, яка задовольняє обмеження за SLA. Саме тому експериментально виміряне зменшення інтегральних витрат ресурсу в цьому підрозділі можна розглядати

як практичне підтвердження реалізації мінімізуючої логіки цільової функції F, введеної в п. 2.1.3.

Слід також зазначити різну природу втрат у двох архітектурах. У статичній схемі спільного хостингу втрати зумовлені тим, що сервер має фіксовану ємність і в моменти локальних піків не може збільшити доступний ресурс. У кластерній архітектурі основні втрати виникають не через відсутність потенціалу масштабування як такого, а в моменти переходу, коли навантаження вже зросло, а нові сервери ще не завершили підключення. Саме ця обставина буде детально досліджена в наступному експерименті, присвяченому реактивному зростанню трафіку та поведінці системи під атакою.

Виконане порівняльне дослідження показало, що для сформованого тижневого профілю навантаження адаптивний кластерний підхід забезпечує практично той самий тижневий рівень обслуговування, що й класична схема спільного хостингу, але потребує приблизно на третину менше інтегральних витрат оперативної пам’яті. Це підтверджує практичну доцільність запропонованої інформаційної технології для середовища масового хостингу слабо навантажених веб-сервісів.

Отримане в цьому підрозділі значення інтегральної економії ресурсів не слід трактувати як універсальну сталу величину для будь-якого середовища вебхостингу. Воно характеризує саме досліджуваний клас тижневих профілів навантаження, сформованих у межах цього експерименту. За інших рівних умов більша нерівномірність трафіку в часі, тобто вираженіші добові та тижневі коливання, створює більший потенціал для консолідації ресурсів у кластерній архітектурі. Навпаки, більш рівномірний профіль навантаження зменшує очікуваний виграш, оскільки потреба в постійно активній ємності стає вищою. Одержаний у цьому підрозділі результат слід розглядати разом із висновками п. 4.5.4, де було показано вплив резервування на втрати в перехідних режимах.

Важливим зауваженням є той факт, що практична реалізація тижневого профілю виконувалась у вигляді кусково-сталого навантаження на інтервалах по 2 хвилини. Це було обумовлено технічними вимогами до відтворюваності довготривалого

експерименту. Унаслідок такої дискретизації перехід між сусідніми рівнями навантаження відбувався стрибкоподібно. Найбільший зафіксований приріст становив 327 RPS між значеннями 1597 RPS та 1924 RPS у середу між 19:36 та 19:38. Попри це, у межах досліджуваного профілю кластер зберіг цільовий рівень якості обслуговування. Отже, двохвилинна дискретизація не спростила умови експерименту, а навпаки, зробила їх жорсткішими. Питання стійкості системи до ще стрімкішого реактивного зростання трафіку, зокрема в умовах атаки, розглянуто в наступному експерименті.

4.7. Дослідження стійкості кластера до HTTP flood-атак та внеску асинхронного контуру захисту

У цьому підрозділі перевіряється стійкість розробленої кластерної архітектури в умовах реактивного зростання навантаження, спричиненого HTTP flood-атакою, а також оцінюється внесок асинхронного контуру захисту в зменшення втрат легітимного трафіку та витрат обчислювальних ресурсів. На відміну від попередніх експериментів, де досліджувались переважно стаціонарні або плавно змінні режими, тут розглядається сценарій різкого додаткового навантаження на один сайт у складі масиву з 10 000 ресурсів.

Логіка експерименту побудована так, щоб спочатку оцінити поведінку кластера без розпізнавання атаки, коли єдиним доступним механізмом реакції є масштабування backend-рівня. Після цього на тих самих рівнях атакувального навантаження досліджується система з увімкненим асинхронним контуром захисту, який передає копії запитів до черги повідомлень, виконує їх пакетний аналіз у модулі ML-Worker, формує рішення про блокування та поширює його до балансувальника через механізм чорного списку. Саме таке порівняння дає змогу окремо оцінити ефект від масштабування і окремо — ефект від раннього відсікання шкідливого трафіку.

Для розділення легітимного та шкідливого трафіку в експерименті використовувався службовий заголовок `X-Test-IP`, за допомогою якого балансувальник підміняв джерельну IP-адресу запиту. Це дало змогу в межах стенда емулювати

надходження запитів від різних клієнтів і відтворити сценарій, у якому бот-трафік походить із заданого пулу адрес. На стороні ML-Worker як шкідливий було визначено трафік із діапазону `1.1.1.1/24`, тоді як усі інші адреси вважалися легітимними. Така постановка допомогла безпосередньо вимірювати саме втрати легітимного трафіку.

Щоб отримати найгірший практично значущий сценарій, базове легітимне навантаження було вибрано на рівні 1124 RPS. Це значення майже збігається з оцінкою середнього трафіку для 10 000 сайтів, але водночас відповідає гранично стійкому режиму для конфігурації з п’яти активних серверів. Така постановка дає змогу досліджувати поведінку кластера саме біля межі його поточної ємності. На сервері генерації навантаження формувалися два потоки: легітимний із рівнем 1124 RPS та бот-трафік із рівнями 500, 1000, 1500 і 2000 RPS.

4.7.1. Час стабілізації кластера та втрати легітимного трафіку без розпізнавання атаки

На першому етапі було досліджено поведінку кластера за відсутності асинхронного контуру захисту. Це означає, що після появи додаткового трафіку система могла реагувати лише шляхом збільшення кількості backend-серверів. Час розгортання нового сервера в експериментальному стенді становив 90 с. За параметрів контуру керування, розглянутих у підрозділі 4.5, консервативна оцінка повного часу реакції кластера дорівнює близько 101 с. Однак у цьому експерименті приріст навантаження відбувався реактивно, протягом однієї-двох секунд. У такій ситуації балансувальник не потребує повного проходження всього вікна згладжування, і тому для помірних рівнів атаки очікуваний час стабілізації оцінювався приблизно в 94—95 с. Для вищих рівнів атаки додаткову роль починала відігравати інерційність генератора навантаження, через яку повний вихід на новий рівень інтенсивності займав дещо більше часу, а отже і період стабілізації зростав.

Для кожного значення бот-трафіку виконувалося по три прогони, після чого в табл. 4.12 наведено середні результати. У цьому експерименті вимірювалися максимальний сумарний потік, час стабілізації кластера та кількість втрачених легітимних запитів.

Таблиця 4.12 - Результати стабілізації кластера за різних масштабів атаки у разі вимкненого контуру захисту.

Інтенсивність атаки	Максимальний сумарний потік	Час стабілізації	Втрачено легітимних запитів
500 RPS	1624 RPS	94 с	27 546
1000 RPS	2124 RPS	94 с	46 415
1500 RPS	2624 RPS	100 с	58 214
2000 RPS	3124 RPS	101 с	66 162

Наведені результати показують, що масштабування саме по собі дійсно дає можливість кластеру зберігати працездатність навіть у разі різкого зростання навантаження. Проте втрати легітимного трафіку в такому режимі тривають увесь час, необхідний для введення в роботу нових серверів. Саме тому навіть за відносно помірної атаки 500 RPS було втрачено понад 27 тис. запитів, а зі зростанням інтенсивності атаки до 2000 RPS втрати зросли до понад 66 тис. запитів. Час стабілізації водночас залишався в діапазоні від 94 до 101 с, що добре узгоджується з параметрами контуру керування, закладеними в математичну модель.

Отримані дані свідчать, що в режимі без розпізнавання атаки кластерна архітектура виграє у статичних схем завдяки можливості нарощувати ємність, але робить це ціною підвищених витрат ресурсів і неминучих втрат легітимного трафіку в перехідному режимі. Інакше кажучи, масштабування в такому сценарії не ліквідує причину перевантаження, а лише поступово пристосовує систему до нового, зокрема шкідливого, рівня навантаження.

4.7.2. Внесок асинхронного контуру захисту в скорочення часу деградації та витрат ресурсів

На другому етапі було досліджено той самий клас атак, але вже з увімкненим асинхронним контуром захисту. Для ML-Worker час оброблення одного пакета запитів

було взято на рівні 45 мс, а накопичувальний поріг спрацьовування встановлено рівним 10 запитам з однієї бот-адреси. Значення затримки відповідає параметру імітатора, обґрунтованому в п. 3.4.4 як реалістична оцінка для легковагового аналітичного модуля. Вибір порога в 10 запитів узгоджується з результатами роботи [114], де показано, що 99 % сесій вебботів класифікуються протягом 6 запитів, а максимальна кількість запитів до прийняття рішення у тестовому наборі становила 19. Тому прийняте значення порогу помірно консервативне і забезпечує компроміс між швидкістю виявлення та надійністю рішення.

Для формування пакетів запитів модуль не використовував жорстко заданий період опитування RabbitMQ. Натомість він працював у безперервному циклі, намагаючись сформувати пакет до 1024 повідомлень шляхом послідовних читань із черги. Коли пакет досягав максимального розміру або черга спорожнювалася, він негайно передавався на оброблення. Якщо черга була порожньою, потік переходив у короткий режим очікування 100 мс, після чого знову робив спробу читання. Таким чином, частота взаємодії з чергою виявлялася адаптивною: за наявності трафіку цикл визначався переважно затримкою аналізу 45 мс, а за відсутності навантаження — паузою 100 мс. Середній розмір сформованого пакету в проведених прогонах становив 134 запити.

Після накопичення достатньої кількості подій для окремих адрес модуль ML-Worker формував рішення про блокування та відправляв адреси у чергу Redis. Балансувальник зчитував адреси з черги та переносив їх до локального чорного списку. Далі запити від уже виявлених бот-адрес відсіювалися до етапу маршрутизації на backend-рівень. Результат тестування наведено в табл. 4.13.

Таблиця 4.13 - Результати стабілізації кластера за різних масштабів атаки у разі увімкненого контуру захисту

Інтенсивність атаки	**Максимальний сумарний потік**	**Час стабілізації**	**Втрачено легітимних запитів**
500 RPS	1624 RPS	15 с	3 814
1000 RPS	2124 RPS	16 с	8 780
1500 RPS	2624 RPS	17 с	13 789

Інтенсивність атаки	Максимальний сумарний потік	Час стабілізації	Втрачено легітимних запитів
2000 RPS	3124 RPS	18 с	18 973

Порівняння табл. 4.12 і 4.13 показує принципову різницю між двома режимами. За вимкненого контуру захисту легітимний трафік втрачався протягом усього часу, потрібного для масштабування кластера. За увімкненого захисту втрати відбувалися лише протягом періоду, необхідного для виявлення шкідливих адрес і поширення рішень про блокування. Саме тому час стабілізації зменшився приблизно з 94-101 секунд до 15—18 секунд, тобто орієнтовно у 5,5—6,3 раза (рис. 4.13). Аналогічно зменшились і втрати легітимного трафіку. Для всіх рівнів атаки скорочення становило приблизно від 4,7 до 5,0 раза.

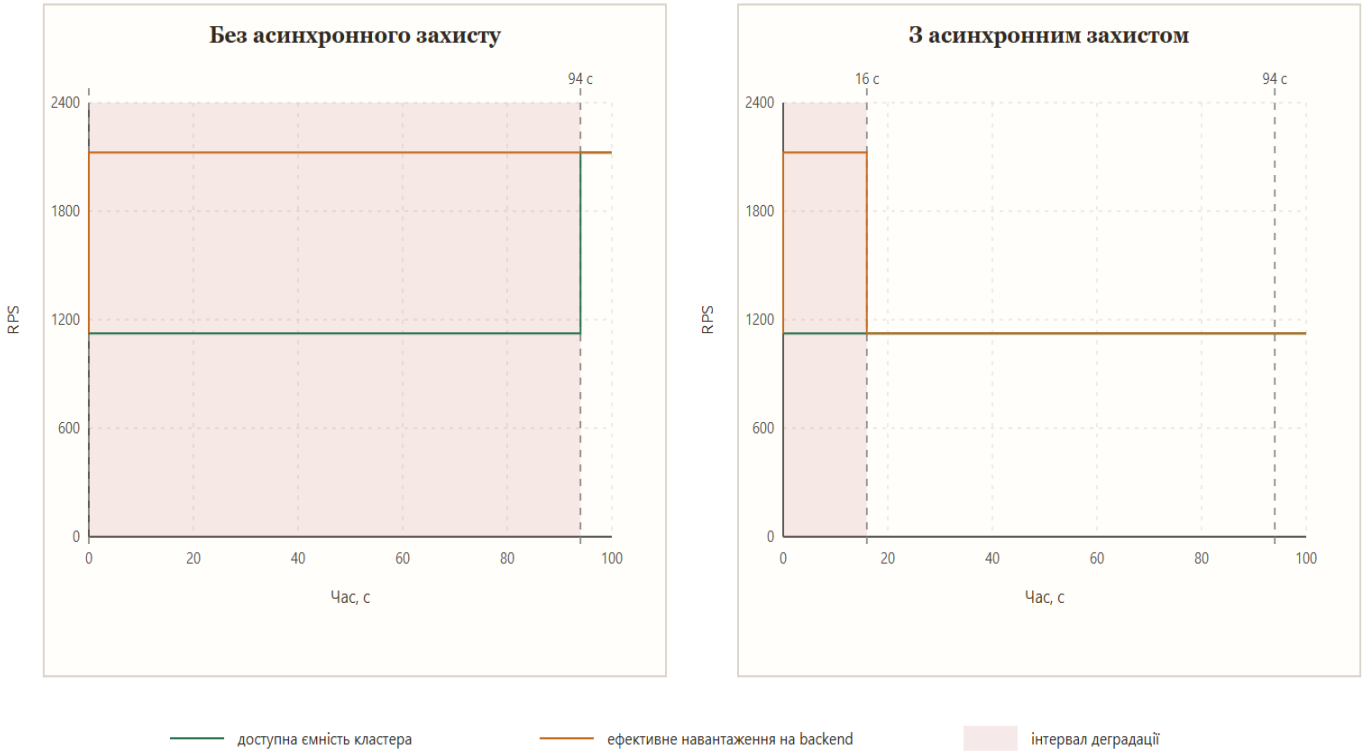

Рисунок 4.13 - Порівняння втрат запитів під час атаки 1000 RPS із захистом та без нього.

У термінах методу, наведеного в підрозділі 2.3, це означає, що практична цінність асинхронного контуру захисту полягає у скороченні часу реакції системи до рівня, за якого блокування шкідливого джерела відбувається раніше, ніж атакувальний потік

встигає критично вичерпати доступну ємність кластера. Саме це пояснює, чому за наявності захисту втрати легітимного трафіку виявляються істотно меншими, ніж у сценарії, де система покладається лише на повільніший контур масштабування.

Отриманий результат має важливу практичну інтерпретацію. Масштабування саме по собі дозволяє системі уникнути повної відмови, однак змушує її витрачати ресурси на обслуговування шкідливого потоку доти, доки нові сервери не будуть введені в експлуатацію. Асинхронний контур захисту працює інакше: він не замінює масштабування, але суттєво скорочує період деградації унаслідок того, що після своєчасного виявлення ознак атаки бот-трафік відсіюється набагато раніше, ніж кластер встигає повністю перебудувати свою ємність. Завдяки цьому зменшується як кількість втрачених легітимних запитів, так і тривалість перебування системи в режимі надмірного ресурсоспоживання.

Цей ефект добре ілюструє рис. 4.14, на якому показано зміну кількості активних серверів під час атаки в обох сценаріях. Без захисту кластер змушений утримувати підвищений рівень ємності протягом усього часу дії шкідливого потоку. За наявності асинхронного контуру захисту кількість активних серверів після виявлення та блокування бот-адрес повертається до рівня, достатнього лише для легітимного трафіку. Повернення до базового масштабу відбувається не миттєво, а з урахуванням логіки охолодження, яка в дослідному прототипі становила 5 хвилин. Саме цим пояснюється часовий зсув між фактичним виявленням та блокуванням атаки системою і моментом остаточного зменшення кількості активних вузлів.

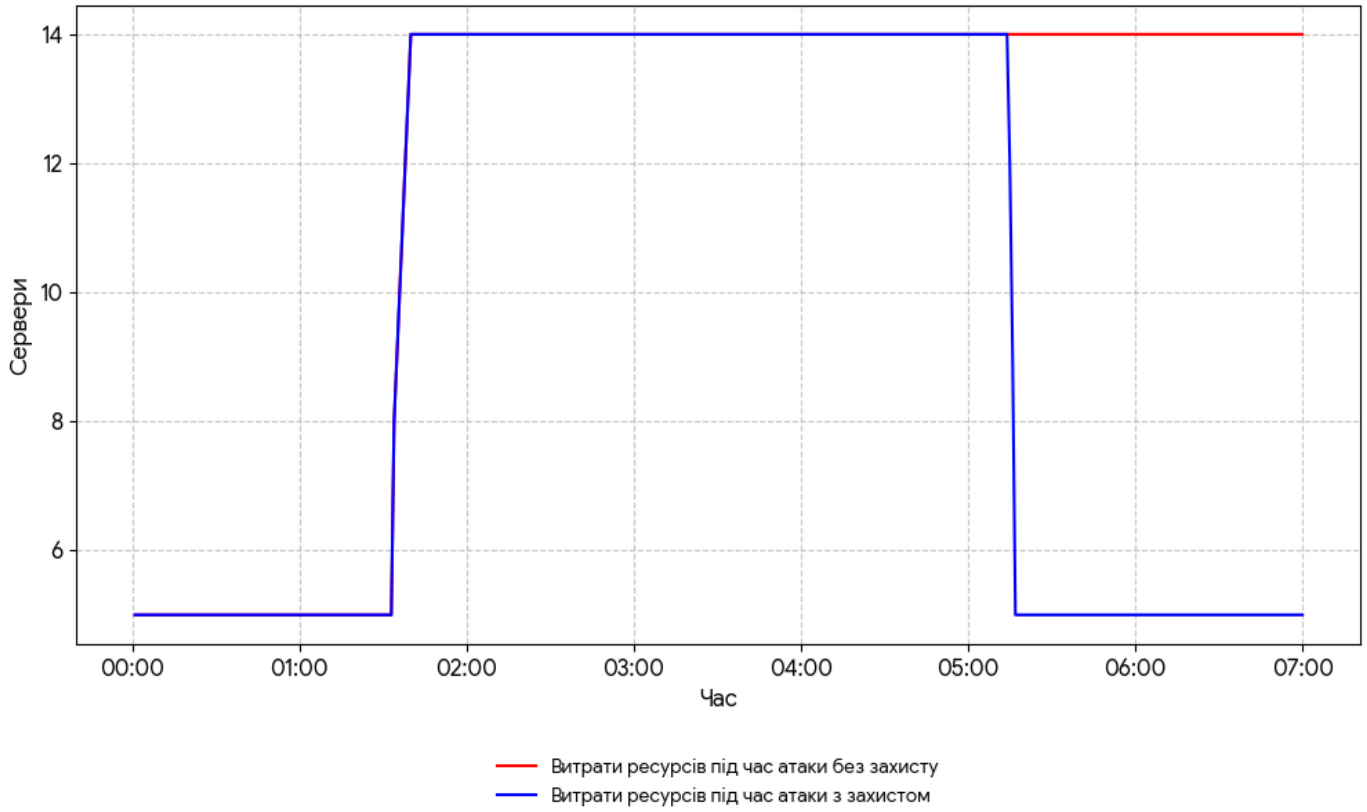

Рисунок 4.14 - Порівняння витрат ресурсів під час атаки за наявності та відсутності асинхронного контуру захисту.

Отже, проведений експеримент показав, що сама лише кластеризація підвищує живучість системи в умовах різкого зростання навантаження, однак не усуває значних втрат легітимного трафіку в період масштабування. Додавання асинхронного контуру захисту принципово змінює цю поведінку: час стабілізації зменшується приблизно у шість разів, втрати легітимного трафіку — приблизно у п’ять разів, а потреба в тривалому утриманні надлишкової кількості серверів істотно знижується. Це підтверджує, що асинхронне виявлення й блокування шкідливих джерел не лише покращує показники доступності, а й підвищує ресурсну ефективність у режимі атаки.

4.8. Висновок до четвертого розділу

У четвертому розділі виконано комплексну експериментальну перевірку запропонованої інформаційної технології в контрольованому середовищі. За результатами практичних експериментів сформульовано такі висновки.

- Експериментально перевірено працездатність програмного комплексу.
- Підтверджено достатність експериментального стенда та відсутність сторонніх «вузьких місць» у межах досліджуваних режимів навантаження.
- Показано, що реалізований балансувальник навантаження здатний стабільно працювати в багатодоменному TLS-середовищі з великою кількістю доменів і сертифікатів, а витрати оперативної пам’яті на підтримку сертифікатного стану мають передбачуваний характер.
- Підтверджено адекватність програмної реалізації модуля прогнозування ємності та механізму масштабування, які забезпечують коректне визначення необхідної кількості обчислювальних вузлів відповідно до поточного навантаження.
- Доведено перевагу адаптивного кластерного підходу над традиційним сумісним хостингом за інтегральним використанням оперативної пам’яті та встановлено, що асинхронний контур аналізу HTTP/HTTPS-запитів підвищує стійкість системи до атак типу HTTP flood.

Отже, результати четвертого розділу підтвердили практичну придатність розробленої інформаційної технології, коректність її програмної реалізації та ефективність запропонованих моделей.

ВИСНОВКИ

У дисертаційній роботі виконано актуальне науково-прикладне завдання розроблення інформаційної технології оптимізації використання обчислювальних ресурсів у хмарних кластерах для динамічного хостингу веб-сервісів яка, на відміну від традиційного статичного розміщення, забезпечує аналітично обґрунтоване планування ємності, адаптивне масштабування та поєднання ресурсної ефективності з прикладним захистом без погіршення швидкодії основного тракту оброблення легітимних запитів.

Проаналізовано сучасні архітектурні підходи до організації масового вебхостингу, методів консолідації ресурсів, масштабування та прикладного захисту. Встановлено, що для середовища масового розміщення слабо навантажених вебресурсів ключовим чинником ефективності є не ізольована оптимізація окремих серверів, а керована консолідація великої кількості стохастичних навантажень у межах спільного хмарного кластера.

Розроблено математичну модель планування ємності хмарного кластера для динамічного вебхостингу, у якій оперативну пам’ять формалізовано як критично обмежувальний ресурс, що визначає допустиму конкурентність оброблення запитів. На основі втратної моделі Ерланга встановлено зв’язок між інтенсивністю вхідного потоку, середнім часом обслуговування, місткістю системи та ймовірністю блокування, що дало змогу перейти від евристичного резервування потужності до аналітичного визначення мінімально достатньої конфігурації кластера.

Розроблено метод асинхронного контуру прикладного захисту, який описує підсистему відкладеного аналізу HTTP/HTTPS-запитів поза критичним шляхом їх оброблення. Метод бере до уваги обмежену чергу, багатоканальне обслуговування, попередню ACL-фільтрацію та затримку верифікаціїщо дає можливість поєднати вимоги безпеки та продуктивності.

Удосконалено метод визначення необхідної конфігурації хмарного кластера за стохастичних коливань навантаження. На відміну від відомих підходів, запропонований метод базується на чисельно стійкому рекурентному обчисленні ймовірності

блокування за формулою Ерланга B, ітеративному доборі мінімально достатньої місткості та застосуванні порогового і часового гістерезису, що зменшує ризик автоколивань і підвищує стійкість керувального контуру.

Набув подальшого розвитку метод кількісного оцінювання ресурсної ефективності консолідації в масовому вебхостингу, який, на відміну від локальних оцінок завантаження окремих вузлів, ґрунтується на інтегральному порівнянні витрат ресурсу в часі для традиційної моделі статичного розміщення та адаптивного хмарного кластера. Це дало змогу кількісно оцінити ефект спільного резерву та статистичного мультиплексування навантаження.

Обґрунтовано архітектурні принципи побудови динамічного хмарного кластера без збереження стану на обчислювальних вузлах, із централізованою керованою точкою входу, винесеним термінуванням TLS, динамічним застосуванням правил доступу, механізмами телеметрії та формуванням керувальних сигналів масштабування. Це забезпечує безшовне додавання і виведення вузлів з кластера та гаряче оновлення конфігурації багатодоменного середовища.

Експериментально підтверджено працездатність запропонованої інформаційної технології в багатодоменному TLS-середовищі з великим числом доменів. Показано, що програмна реалізація зберігає стабільність під навантаженням, підтримує гаряче оновлення конфігурації та характеризується передбачуваним зростанням витрат оперативної пам’яті на підтримку сертифікатного стану. Встановлено, що поєднання динамічного масштабування з асинхронним контуром прикладного захисту підвищує стійкість системи до атак типу HTTP flood, скорочує час стабілізації роботи кластера та зменшує втрати легітимного трафіку. Це підтверджує доцільність винесення ресурсоємного аналізу запитів за межі критичного шляху оброблення.

Практичне значення одержаних результатів полягає в тому, що результати дисертаційної роботи впроваджено в ТОВ «АксонСофт» для аудиту поточної ефективності використання серверного парку компанії та в навчальному процесі Державного університету «Київський авіаційний інститут».

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Chizhov A., Fesenko A. Web hosting companies' client solutions: A study of a strategic standpoint [Special issue]. Corporate & Business Strategy Review. 2025. Vol. 6, No. 1. P. 421–429. DOI: 10.22495/cbsrv6i1siart18.
2. Netcraft. January 2026 Web Server Survey. 2026. URL: https://www.netcraft.com/blog/january-2026-web-server-survey (дата звернення: 30.03.2026).
3. W3Techs. Usage statistics and market shares of content management systems. 2026. URL: https://w3techs.com/technologies/overview/content_management (дата звернення: 30.03.2026).
4. Crovella M. E., Bestavros A. Self-similarity in World Wide Web traffic: evidence and possible causes. IEEE/ACM Transactions on Networking. 1997. Vol. 5, No. 6. P. 835–846. DOI: 10.1109/90.650143.
5. Xavier H. S. The Web Unpacked: A Quantitative Analysis of Global Web Usage. Proceedings of the 20th International Conference on Web Information Systems and Technologies (WEBIST 2024). 2024. P. 183–190. DOI: 10.5220/0012905900003825.
6. Huss N. How Many Websites Are There in the World? Siteefy. URL: https://siteefy.com/how-many-websites-are-there/ (дата звернення: 30.03.2026).
7. Verisign. Domain Name Industry Brief Quarterly Report: Q4 2025. 2026. URL: https://blog.verisign.com/domain-names/q4-2025-domain-name-industry-brief-quarterly-report/ (дата звернення: 22.02.2026).
8. Чижов О., Фесенко А. Метод планування потужностей для консолідації ресурсів у масовому хмарному веб-хостингу. *Проблеми інформатизації та управління*. 2025. Т. 83, № 4. С. 151–161. DOI: 10.18372/2073-4751.84.20905.
9. Web caching and Zipf-like distributions: evidence and implications / L. Breslau, P. Cao, L. Fan [et al.]. Proceedings IEEE INFOCOM '99. 1999. Vol. 1. P. 126–134. DOI: 10.1109/INFCOM.1999.749260.

10. Defining multi-tenancy: A systematic mapping study on the academic and the industrial perspective / J. Kabbedijk, C.-P. Bezemer, S. Jansen [et al.]. Journal of Systems and Software. 2015. Vol. 100. P. 139–148. DOI: 10.1016/j.jss.2014.10.034.
11. Fortune Business Insights. Web Hosting Services Market Size, Share & Industry Analysis, By Type (Shared Hosting, Dedicated Hosting, Collocated Hosting, Virtual Private Server Hosting, Managed Hosting, Self-Managed Hosting, and Others), By Application, By Deployment, and Regional Forecast, 2026–2034. 2026. URL: https://www.fortunebusinessinsights.com/industry-reports/web-hosting-services-market-100863 (дата звернення: 30.03.2026).
12. Zahid E., Lazaravičius S. Web hosting statistics 2026: Key trends, facts & global insights. Hostinger Tutorials. 2025. URL: https://www.hostinger.com/uk/tutorials/web-hosting-statistics (дата звернення: 30.03.2026).
13. Banga G., Druschel P., Mogul J. C. Resource containers: A new facility for resource management in server systems. Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI ’99). 1999. P. 45–58. DOI: 10.5555/296806.296810.
14. Harchol-Balter M. Performance Modeling and Design of Computer Systems: Queueing Theory in Action. Cambridge : Cambridge University Press, 2013. 576 p. DOI: 10.1017/CBO9781139226424.
15. Singleton D. Website Performance Monitoring. Software Quality and Software Testing in Internet Times / eds. D. Meyerhoff, B. Laibarra, R. van der Pouw Kraan, A. Wallet. Berlin ; Heidelberg : Springer-Verlag, 2002. P. 199–216. DOI: 10.1007/978-3-642-56333-1_13.
16. Cloud computing synopsis and recommendations : NIST SP 800-146 / M. L. Badger, T. Grance, R. Patt-Corner, J. Voas. Gaithersburg, MD : National Institute of Standards and Technology, 2012. DOI: 10.6028/NIST.SP.800-146.
17. Arapakis I., Bai X., Cambazoglu B. B. Impact of response latency on user behavior in web search. Proceedings of the 37th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR ’14). New York, NY : ACM, 2014. P. 103–112. DOI: 10.1145/2600428.2609627.

18. Milliseconds make millions: A study on how improvements in mobile site speed positively affect a brand’s bottom line. Google; Deloitte Ireland LLP, 2019. URL: https://www.thinkwithgoogle.com/_qs/documents/9757/Milliseconds_Make_Millions_report_hQYAbZJ.pdf (дата звернення: 30.03.2026).
19. Google. HTTPS encryption on the web. Google Transparency Report. 2026. URL: https://transparencyreport.google.com/https/overview (дата звернення: 30.03.2026).
20. Google. HTTPS FAQs. Transparency Report Help Center. Б. р. URL: https://support.google.com/transparencyreport/answer/7381231?hl=uk (дата звернення: 30.03.2026).
21. The Cost of the “S” in HTTPS / D. Naylor, A. Finamore, I. Leontiadis [et al.]. Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies (CoNEXT ’14). 2014. P. 133–140. DOI: 10.1145/2674005.2674991.
22. Pollard B., Wagner J. Optimize Time to First Byte. web.dev. 2023. URL: https://web.dev/articles/optimize-ttfb (дата звернення: 30.03.2026).
23. Jennings B., Stadler R. Resource Management in Clouds: Survey and Research Challenges. Journal of Network and Systems Management. 2015. Vol. 23, No. 3. P. 567–619. DOI: 10.1007/s10922-014-9307-7.
24. Tanenbaum A. S., Bos H. Modern Operating Systems. 4th ed. Boston : Pearson, 2014. 1136 p.
25. Red Hat. Managing Out of Memory states. Red Hat Documentation. Б. р. URL: https://docs.redhat.com/en/documentation/red_hat_enterprise_linux_for_real_time/8/html/optimizing_rhel_8_for_real_time_for_low_latency_operation/assembly_managing-out-of-memory-states_optimizing-rhel8-for-real-time-for-low-latency-operation (дата звернення: 30.03.2026).
26. A view of cloud computing / M. Armbrust, A. Fox, R. Griffith [et al.]. Communications of the ACM. 2010. Vol. 53, No. 4. P. 50–58. DOI: 10.1145/1721654.1721672.

27. Google Cloud. Take advantage of elasticity. Cloud Architecture Center. 2024. URL: https://docs.cloud.google.com/architecture/framework/performance-optimization/elasticity (дата звернення: 30.03.2026).
28. Xie Y., Yu S.-Z. Monitoring the Application-Layer DDoS Attacks for Popular Websites. IEEE/ACM Transactions on Networking. 2009. Vol. 17, No. 1. P. 15–25. DOI: 10.1109/TNET.2008.925628.
29. Durumeric Z., Bailey M., Halderman J. A. An Internet-Wide View of Internet-Wide Scanning. 23rd USENIX Security Symposium (USENIX Security 14). 2014. P. 65–78. DOI: 10.5555/2671225.2671230.
30. Cloud shared hosting DDoS resistance and potential ways of protection. A. Chizhov, A. Fesenko, V. Ziuziun, D. Basshykyzy. CEUR Workshop Proceedings. 2025. Vol. 3925. P. 13–23. URL: https://ceur-ws.org/Vol-3925/paper02.pdf (дата звернення: 30.03.2026).
31. Cloudflare. DDoS threat report for 2025 Q4. Cloudflare Radar Reports. 2026. URL: https://radar.cloudflare.com/reports/ddos-2025-q4 (дата звернення: 30.03.2026).
32. OWASP Foundation. OAT-007 Credential Cracking. OWASP Automated Threats to Web Applications. Б. р. URL: https://owasp.org/www-project-automated-threats-to-web-applications/assets/oats/EN/OAT-007_Credential_Cracking (дата звернення: 30.03.2026).
33. Jung J., Krishnamurthy B., Rabinovich M. Flash crowds and denial of service attacks: characterization and implications for CDNs and Web sites. Proceedings of the 11th International Conference on World Wide Web (WWW ’02). New York, NY : ACM, 2002. P. 293–304. DOI: 10.1145/511446.511485.
34. Scarfone K. A., Mell P. M. Guide to Intrusion Detection and Prevention Systems (IDPS) : NIST SP 800-94. Gaithersburg, MD : National Institute of Standards and Technology, 2007. DOI: 10.6028/NIST.SP.800-94.
35. Спосіб захисту вебдодатків на основі асинхронного аналізу запитів: пат. 160974 U Україна: H04W 12/08, G06F 21/00, G06F 12/14. Дуднік А. С., Фесенко А. О., Чижов

О. В.; заявник і патентовласник ПрАТ “ВНЗ”МАУП”. № u 2025 02265; заявл. 13.05.2025; опубл. 22.10.2025, Бюл. № 43. 4 с.

36. Xen and the art of virtualization / P. Barham, B. Dragovic, K. Fraser [et al.]. Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP ’03). 2003. P. 164–177. DOI: 10.1145/945445.945462.
37. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors / S. Soltesz, H. Pötzl, M. E. Fiuczynski [et al.]. Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys ’07). New York, NY : Association for Computing Machinery, 2007. P. 275–287. DOI: 10.1145/1272996.1273025.
38. Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments / M. G. Xavier, M. V. Neves, F. D. Rossi [et al.]. 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP). 2013. P. 233–240. DOI: 10.1109/PDP.2013.41.
39. Merkel D. Docker: Lightweight Linux Containers for Consistent Development and Deployment. Linux Journal. 2014. Vol. 2014, No. 239. Art. 2. URL: https://www.linuxjournal.com/content/docker-lightweight-linux-containers-consistent-development-and-deployment (дата звернення: 30.03.2026).
40. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis / C. Reiss, A. Tumanov, G. R. Ganger [et al.]. Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC ’12) (San Jose, CA, USA, October 14–17, 2012). New York, NY : Association for Computing Machinery, 2012. Art. 7. 13 p. DOI: 10.1145/2391229.2391236.
41. Kubernetes. Ingress. Kubernetes Documentation. 2025. URL: https://kubernetes.io/docs/concepts/services-networking/ingress/ (дата звернення: 30.03.2026).

42. F5 NGINX. The design of NGINX Ingress Controller. NGINX Documentation. 2026. URL: https://docs.nginx.com/nginx-ingress-controller/overview/design/ (дата звернення: 30.03.2026).
43. kubernetes/ingress-nginx. All ingress controllers restart at the same time on configuration change. Issue #11083. GitHub. 2024. URL: https://github.com/kubernetes/ingress-nginx/issues/11083 (дата звернення: 30.03.2026).
44. HashiCorp. HashiCorp 2024 State of Cloud Strategy Survey: Connecting cloud maturity to business success. 2024. URL: https://www.hashicorp.com/en/state-of-the-cloud (дата звернення: 30.03.2026).
45. Kubernetes. Overview. Kubernetes Documentation. Б. р. URL: https://kubernetes.io/docs/concepts/overview/ (дата звернення: 30.03.2026).
46. Kubernetes. Deployments. Kubernetes Documentation. Б. р. URL: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/ (дата звернення: 30.03.2026).
47. Kubernetes. Horizontal Pod Autoscaling. Kubernetes Documentation. Б. р. URL: https://kubernetes.io/docs/concepts/workloads/autoscaling/horizontal-pod-autoscale/ (дата звернення: 30.03.2026).
48. Medel V., Tolosana-Calasanz R., Bañares J. Á., Arronategui U., Rana O. F. Characterising resource management performance in Kubernetes. Computers & Electrical Engineering. 2018. Vol. 68. P. 286–297. DOI: 10.1016/j.compeleceng.2018.03.041.
49. HAProxy. Manage SSL certificates. HAProxy Enterprise Documentation. Б. р. URL: https://www.haproxy.com/documentation/haproxy-enterprise/administration/manage-ssl-certs/ (дата звернення: 30.03.2026).
50. Google Cloud. Using Google-managed SSL certificates. Google Kubernetes Engine Documentation. 2026. URL: https://docs.cloud.google.com/kubernetes-engine/docs/how-to/managed-certs (дата звернення: 30.03.2026).
51. Kozlov D. How we ensure Cloudflare customers aren’t affected by Let’s Encrypt’s certificate chain change. Cloudflare Blog. 2024. URL:

https://blog.cloudflare.com/shortening-lets-encrypt-change-of-trust-no-impact-to-cloudflare-customers/ (дата звернення: 30.03.2026).

52. Kubernetes Ingress-NGINX. TLS/HTTPS. Ingress-NGINX Controller Documentation. Б. р. URL: https://kubernetes.github.io/ingress-nginx/user-guide/tls/ (дата звернення: 30.03.2026).
53. Cloud Programming Simplified: A Berkeley View on Serverless Computing / E. Jonas, J. Schleier-Smith, V. Sreekanti [et al.]. Berkeley : University of California, 2019. (Technical Report No. UCB/EECS-2019-3). URL: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.pdf (дата звернення: 30.03.2026).
54. Rise of the Planet of Serverless Computing: A Systematic Review / J. Wen, Z. Chen, X. Jin [et al.]. ACM Transactions on Software Engineering and Methodology. 2023. Vol. 32, No. 5. Art. 131. P. 131:1–131:61. DOI: 10.1145/3579643.
55. Peeking Behind the Curtains of Serverless Platforms / L. Wang, M. Li, Y. Zhang [et al.]. Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC ’18). 2018. P. 133–146. URL: https://www.usenix.org/system/files/conference/atc18/atc18-wang-liang.pdf (дата звернення: 30.03.2026).
56. Amazon Web Services. AWS Lambda Pricing. AWS Documentation. Б. р. URL: https://aws.amazon.com/lambda/pricing/ (дата звернення: 30.03.2026).
57. Razumov P. What does the PMEM LVE limit affect. CloudLinux Knowledge Base. 2024. URL: https://cloudlinux.zendesk.com/hc/en-us/articles/5503038402844-What-does-the-PMEM-LVE-limit-affect (дата звернення: 30.03.2026).
58. Chizhov A., Fesenko A. Intelligent load balancing management in cloud web hosting: Evaluation criteria and methodology. Problems of Informatization and Management. 2025. No. 3(83). P. 138–147. DOI: 10.18372/2073-4751.83.20551.
59. Методи оптимізації розподілу навантаження на обчислювальний ресурс інфраструктури хмарного сервісу. Чижов О. В., Фесенко А. О., Пустосвіт М. С.,

Німченко Т. В. *Захист інформації*. 2023. Т. 25, № 4. С. 207–213. DOI: 10.18372/2410-7840.25.18226.

60. Beloglazov A., Abawajy J., Buyya R. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. Future Generation Computer Systems. 2012. Vol. 28, No. 5. P. 755–768. DOI: 10.1016/j.future.2011.04.017.
61. Kubernetes. Autoscaling Workloads. Kubernetes Documentation. 2025. URL: https://kubernetes.io/docs/concepts/workloads/autoscaling/ (дата звернення: 30.03.2026).
62. Kubernetes. Vertical Pod Autoscaling. Kubernetes Documentation. 2026. URL: https://kubernetes.io/docs/concepts/workloads/autoscaling/vertical-pod-autoscale/ (дата звернення: 30.03.2026).
63. Qu C., Calheiros R. N., Buyya R. Auto-Scaling Web Applications in Clouds: A Taxonomy and Survey. ACM Computing Surveys. 2018. Vol. 51, No. 4. Art. 73. 33 p. DOI: 10.1145/3148149.
64. cPanel. Overselling. cPanel & WHM Documentation. 2019. URL: https://docs.cpanel.net/knowledge-base/cpanel-product/overselling/ (дата звернення: 30.03.2026).
65. Ulrich M. Addressing Cascading Failures. Site Reliability Engineering (Google SRE Book). O'Reilly Media, 2017. URL: https://sre.google/sre-book/addressing-cascading-failures/ (дата звернення: 30.03.2026).
66. Mell P., Grance T. The NIST Definition of Cloud Computing. Gaithersburg, MD : National Institute of Standards and Technology, 2011. (NIST Special Publication 800-145). DOI: 10.6028/NIST.SP.800-145.
67. Kleinrock L. Queueing Systems. Volume I: Theory. New York : Wiley-Interscience, 1975. 448 p. ISBN 9780471491101.
68. ITU-T. Recommendation E.529 (05/97). Network dimensioning using end-to-end GOS objectives. Geneva : International Telecommunication Union, 1997. URL:

https://www.itu.int/rec/dologin_pub.asp?id=T-REC-E.529-199705-I!!PDF-E&lang=s&type=items (дата звернення: 30.03.2026).

69. ISO/IEC 7498-1:1994. Information technology — Open Systems Interconnection — Basic Reference Model: The Basic Model. 2nd ed. 1994. URL: https://www.ecma-international.org/wp-content/uploads/s020269e.pdf (дата звернення: 30.03.2026).
70. Eastlake D. Transport Layer Security (TLS) Extensions: Extension Definitions. RFC 6066. 2011. IETF. URL: https://datatracker.ietf.org/doc/html/rfc6066 (дата звернення: 30.03.2026).
71. Langley A. Overclocking SSL. ImperialViolet. 2010. URL: https://www.imperialviolet.org/2010/06/25/overclocking-ssl.html (дата звернення: 30.03.2026).
72. Barnes R., Hoffman-Andrews J., McCarney D., Kasten J. Automatic Certificate Management Environment (ACME). RFC 8555. 2019. IETF. DOI: 10.17487/RFC8555.
73. Cormen T. H., Leiserson C. E., Rivest R. L., Stein C. Introduction to Algorithms. 4th ed. Cambridge, MA : The MIT Press, 2022. 1312 p. ISBN 9780262046305. URL: https://mitpress.mit.edu/9780262046305/introduction-to-algorithms/ (дата звернення: 30.03.2026).
74. Herlihy M., Shavit N. The Art of Multiprocessor Programming, Revised Reprint. 1st ed. Morgan Kaufmann (Elsevier), 2012. ISBN 978-0-12-397337-5. URL: https://shop.elsevier.com/books/the-art-of-multiprocessor-programming-revised-reprint/herlihy/978-0-12-397337-5 (дата звернення: 30.03.2026).
75. Johansson A. HTTP Load Balancing Performance Evaluation of HAProxy, NGINX, Traefik and Envoy with the Round-Robin Algorithm. Independent thesis (Bachelor level). Skövde: University of Skövde, School of Informatics, 2022. URL: https://www.diva-portal.org/smash/get/diva2:1678660/FULLTEXT01.pdf (дата звернення: 30.03.2026).
76. Bradner S., McQuaid J. Benchmarking Methodology for Network Interconnect Devices. RFC 2544. 1999. IETF. URL: https://www.rfc-editor.org/rfc/rfc2544.html (дата звернення: 30.03.2026).

77. F5. Directives. NGINX Documentation: F5 WAF for NGINX. URL: https://docs.nginx.com/waf/policies/directives/ (дата звернення: 30.03.2026).
78. OWASP Top 10 Team. OWASP Top 10:2021. A07:2021 – Identification and Authentication Failures. OWASP Foundation. 2021. URL: https://owasp.org/Top10/2021/A07_2021-Identification_and_Authentication_Failures/ (дата звернення: 30.03.2026).
79. F5 NGINX. Limiting Access to Proxied HTTP Resources. NGINX Documentation. URL: https://docs.nginx.com/nginx/admin-guide/security-controls/controlling-access-proxied-http/ (дата звернення: 30.03.2026).
80. OWASP Foundation. OWASP ModSecurity Core Rule Set (OWASP CRS). URL: https://owasp.org/www-project-modsecurity-core-rule-set/ (дата звернення: 30.03.2026).
81. Cloudflare. Cloudflare Turnstile (smart CAPTCHA alternative). Cloudflare Docs. 2026. URL: https://developers.cloudflare.com/turnstile/ (дата звернення: 30.03.2026).
82. The Spamhaus Project. Do Not Route Or Peer Lists (DROP). URL: https://www.spamhaus.org/blocklists/do-not-route-or-peer/ (дата звернення: 30.03.2026).
83. Herley C. Automated Detection of Automated Traffic. 31st USENIX Security Symposium (USENIX Security 22) (Boston, MA, USA, August 10–12, 2022). Berkeley, CA : USENIX Association, 2022. P. 1615–1632. ISBN 978-1-939133-31-1. URL: https://www.usenix.org/system/files/sec22-herley.pdf (дата звернення: 30.03.2026).
84. Gajewski M., Hryniewicz O., Jastrzębska A. та ін. Data-driven human and bot recognition from web activity logs based on hybrid learning techniques. Digital Communications and Networks. 2024. Vol. 10, no. 4. P. 1178–1188. DOI: 10.1016/j.dcan.2023.01.020.
85. PHP Group. PHP: Configuration - Manual. PHP Manual. URL: https://www.php.net/manual/en/install.fpm.configuration.php (дата звернення: 30.03.2026).
86. Fielding R., Nottingham M., Reschke J. HTTP Semantics. RFC 9110. 2022. IETF. DOI: 10.17487/RFC9110.

87. Hudson R. Getting to Go: The Journey of Go’s Garbage Collector. The Go Blog. 2018. URL: https://go.dev/blog/ismmkeynote (дата звернення: 30.03.2026).
88. Oracle. Garbage Collector Ergonomics. Java SE 8 Documentation. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gc-ergonomics.html (дата звернення: 30.03.2026).
89. Python Software Foundation. gc — Garbage Collector interface. Python 3 Documentation. URL: https://docs.python.org/3/library/gc.html (дата звернення: 30.03.2026).
90. Rust Project. Understanding Ownership. The Rust Programming Language. URL: https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html (дата звернення: 30.03.2026).
91. Matsakis N. D., Klock F. S., II. The rust language. ACM SIGAda Ada Letters. 2014. Vol. 34, No. 3. P. 103–104. DOI: 10.1145/2692956.2663188.
92. Tokio. Crate tokio. docs.rs. URL: https://docs.rs/tokio/latest/tokio/ (дата звернення: 30.03.2026).
93. Amazon Web Services. What is Amazon VPC? Amazon VPC User Guide. URL: https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html (дата звернення: 30.03.2026).
94. Lindem A., Dogra A. Virtual Router Redundancy Protocol (VRRP) Version 3 for IPv4 and IPv6. RFC 9568. 2024. IETF. DOI: 10.17487/RFC9568.
95. Mockapetris P. Domain Names - Concepts and Facilities. RFC 1034. 1987. IETF. DOI: 10.17487/RFC1034.
96. Sikeridis D., Kampanakis P., Devetsikiotis M. Post-Quantum Authentication in TLS 1.3: A Performance Study. Network and Distributed System Security Symposium. 2020. DOI: 10.14722/ndss.2020.24203.
97. Rust Project. Module sync. The Rust Standard Library. URL: https://doc.rust-lang.org/std/sync/ (дата звернення: 30.03.2026).

98. Linux kernel documentation. What is RCU? – “Read, Copy, Update”. The Linux Kernel documentation. URL: https://docs.kernel.org/RCU/whatisRCU.html (дата звернення: 30.03.2026).
99. Redis. Pub/sub. Redis Documentation. URL: https://redis.io/docs/latest/develop/pubsub/ (дата звернення: 30.03.2026).
100. Redis. EXPIRE. Redis Documentation. URL: https://redis.io/docs/latest/commands/expire/ (дата звернення: 30.03.2026).
101. Rust Project. Module atomic. The Rust Standard Library. URL: https://doc.rust-lang.org/std/sync/atomic/ (дата звернення: 30.03.2026).
102. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional. 1994. 416 p. ISBN 978-0-201-63361-0.
103. Detection of HTTP DDoS Attacks Using NFStream and TensorFlow. Chovanec M., Hasin M., Havrilla M., Chovancová E. Applied Sciences. 2023. Vol. 13, No. 11. Article 6671. DOI: 10.3390/app13116671.
104. LUCID: A Practical, Lightweight Deep Learning Solution for DDoS Attack Detection. Doriguzzi-Corin R., Millar S., Scott-Hayward S., Martinez-del-Rincon J., Siracusa D. IEEE Transactions on Network and Service Management. 2020. Vol. 17, No. 2. P. 876–889. DOI: 10.1109/TNSM.2020.2971776.
105. Lucz G., Forstner B. A Thirty-Day Dataset of Malicious HTTP Requests Blocked by OWASP ModSecurity on a Production Web Server. Data. 2025. Vol. 10, No. 11. Article 186. DOI: 10.3390/data10110186.
106. RabbitMQ. Consumer Acknowledgements and Publisher Confirms. RabbitMQ Documentation. URL: https://www.rabbitmq.com/docs/confirms (дата звернення: 30.03.2026).
107. RabbitMQ. Persistence Configuration. RabbitMQ Documentation. URL: https://www.rabbitmq.com/docs/persistence-conf (дата звернення: 30.03.2026).

108. RabbitMQ. Queue Length Limit. RabbitMQ Documentation. URL: https://www.rabbitmq.com/docs/maxlength (дата звернення: 30.03.2026).
109. RabbitMQ. Time-To-Live and Expiration. RabbitMQ Documentation. URL: https://www.rabbitmq.com/docs/ttl (дата звернення: 30.03.2026).
110. NVIDIA. Best Practices. NVIDIA TensorRT. NVIDIA Documentation. URL: https://docs.nvidia.com/deeplearning/tensorrt/latest/performance/best-practices.html (дата звернення: 30.03.2026).
111. Kerrisk M. The Linux Programming Interface: A Linux and UNIX System Programming Handbook. San Francisco: No Starch Press. 2010. 1552 p.
112. Yang J., Leskovec J. Patterns of Temporal Variation in Online Media. Proceedings of the Fourth ACM International Conference on Web Search and Data Mining (WSDM ’11). New York: ACM. 2011. P. 177–186. DOI: 10.1145/1935826.1935863.
113. Web Traffic Time Series Forecasting Using LSTM Neural Networks with Distributed Asynchronous Training. Casado-Vara R., Martin del Rey A., Pérez-Palau D., de-la-Fuente-Valentín L., Corchado J. M. Mathematics. 2021. Vol. 9, No. 4. Article 421. DOI: 10.3390/math9040421.
114. Efficient on-the-fly Web bot detection. Suchacka G., Cabri A., Rovetta S., Masulli F. Knowledge-Based Systems. 2021. Vol. 223. Article 107074. DOI: 10.1016/j.knosys.2021.107074.

ДОДАТОК 1. КЛЮЧОВІ ЕЛЕМЕНТИ МОДУЛЯ PREDICTION БАЛАНСУВАЛЬНИКА (ILB)

```
    // Функція підтримує ковзне часове вікно значень RPS сталої довжини.
    // Після заповнення вікна кожне нове значення витісняє найстаріше, що
    // забезпечує безперервне оновлення вибірки для наступного циклу прогнозу.
    fn push_rps_sample(&self, rps: u64) -> bool {
        let mut window = self.window.write().unwrap();
        if window.len() == self.config.sliding_window {
            window.pop_front();
        }
        window.push_back(rps);
        window.len() == self.config.sliding_window
    }


    fn recompute(&self, cluster: &ClusterState, scaling: &Arc<ScalingState>) {
        let first_recomputation = self.total_recomputation_cycles.load(Ordering::Acquir
e) == 0;
        let started_at = Instant::now();
        let smoothed_rps = {
            let window = self.window.read().unwrap();
            let sum: u64 = window.iter().copied().sum();
            sum as f64 / self.config.sliding_window as f64
        };
        let average_request_time_secs = avg_request_time_secs();
        let offered_traffic = smoothed_rps * average_request_time_secs;


        let available_slots_per_server =
            self.cluster_config.backend_useful_memory_mib / self.config.memory_unit_mi
b;
        let required_slots = required_slots(
            offered_traffic,
            target_blocking_probability(self.config.sla),
        );
        let calculated_servers = ceil_div(required_slots, available_slots_per_server);
```

```
        let required_servers = calculated_servers
            .max(self.config.min_servers)
            .min(self.config.max_servers);

        let current_servers = cluster.backend_count();
        let pending_servers = scaling.pending_servers();
        let effective_servers = current_servers + pending_servers;

        let mut predicted_delta = required_servers as i32 - effective_servers as i32;
        if predicted_delta < 0 {
            // Для сценарію scale-in перевіряється, чи не призведе вилучення
            // надлишкових серверів до перевищення допустимої ймовірності
            // блокування. З цією метою для цільової конфігурації після
            // зменшення кластера обчислюється Erlang B, а допустимий поріг
            // додатково посилюється коефіцієнтом reserve-ratio, що формує
            // резерв пропускної здатності відносно базової SLA-вимоги.
            let scale_in_target_probability =
                target_blocking_probability(self.config.sla) * (1.0 - self.config.reser
ve_ratio);
            let slots_after_scale_in = required_servers * available_slots_per_server;
            let block_probability = erlang_b(offered_traffic, slots_after_scale_in);
            if block_probability > scale_in_target_probability {
                predicted_delta = 0;
            }
        }

        let effective_delta = if predicted_delta > 0 {
            self.reset_scale_down_cooldown();
            scaling.submit_scale(predicted_delta);
            predicted_delta
        } else if predicted_delta == 0 {
            self.reset_scale_down_cooldown();
            0
        } else {
```

```
        self.apply_scale_down_cooldown(predicted_delta, scaling)
    };

    if first_recomputation {
        scaling.schedule_configured_start_delay(
            self.cluster_config.backend_start_time_sec,
            STARTUP_DELAY_SWITCH_GRACE_SEC,
        );
    }

    let duration_ms = started_at.elapsed().as_millis() as u64;
    self.last_recomputation_duration_ms
        .store(duration_ms, Ordering::Release);
    self.total_recomputation_cycles
        .fetch_add(1, Ordering::Relaxed);
    self.last_smoothed_rps_bits
        .store(smoothed_rps.to_bits(), Ordering::Release);
    self.last_offered_traffic_bits
        .store(offered_traffic.to_bits(), Ordering::Release);
    self.last_available_slots_per_server
        .store(available_slots_per_server, Ordering::Release);
    self.last_required_slots
        .store(required_slots, Ordering::Release);
    self.last_computed_servers
        .store(required_servers, Ordering::Release);
    self.last_servers_delta
        .store(effective_delta, Ordering::Release);
    update_max_u64(&self.max_recomputation_duration_ms, duration_ms);
    update_min_u64(&self.min_recomputation_duration_ms, duration_ms);
}

// Механізм cooldown для scale-in передбачає затримку між виявленням
// потенційної надлишковості ресурсів
fn apply_scale_down_cooldown(
```

```
        &self,
        predicted_delta: i32,
        scaling: &Arc<ScalingState>,
    ) -> i32 {
        let cooldown_sec = self.cluster_config.cooldown_sec;
        let now = Instant::now();
        let mut cooldown = self.scale_down_cooldown.lock().unwrap();

        match cooldown.as_mut() {
            None => {
                *cooldown = Some(ScaleDownCooldownState {
                    started_at: now,
                    base_delta: predicted_delta,
                });
                log_info(format!(
                    "prediction scale-down cooldown started: base_delta={} cooldown_sec
={}",
                    predicted_delta, cooldown_sec
                ));
                0
            }
            Some(state) => {
                if predicted_delta > state.base_delta {
                    state.base_delta = predicted_delta;
                    log_info(format!(
                        "prediction scale-down cooldown updated: base_delta={} cooldown
_sec={}",
                        state.base_delta, cooldown_sec
                    ));
                }

                if state.started_at.elapsed().as_secs() >= cooldown_sec {
                    let delta_to_send = state.base_delta;
                    *cooldown = None;
```

```
                scaling.submit_scale(delta_to_send);
                log_info(format!(
                    "prediction scale-down cooldown expired: sending delta={}",
                    delta_to_send
                ));
                delta_to_send
            } else {
                0
            }
        }
    }
}

// offered_traffic в Ерлангах є середнім стохастичним навантаженням, а не
// детермінованою кількістю одночасно активних запитів. Тому навіть при
// offered_traffic = channels імовірність блокування загалом залишається
// додатною, а мінімальна кількість каналів визначається ітераційним пошуком
// за умовою Erlang B: B(A, m) <= P_target.
fn required_slots(offered_traffic: f64, target_probability: f64) -> usize {
    if offered_traffic <= 0.0 {
        return 0;
    }

    let mut channels = offered_traffic.ceil() as usize;
    loop {
        let probability = erlang_b(offered_traffic, channels);
        if probability <= target_probability {
            return channels;
        }
        channels += 1;
    }
}

// У прототипному стенді середній час обслуговування запиту приймається сталим
```

```rust
// і рівним 1 с, що дає змогу спростити реалізацію без втрати узгодженості з
// експериментальною моделлю, де тестові backend-вузли повертають відповідь із
// фіксованою затримкою.
fn avg_request_time_secs() -> f64 {
    1.0
}

// Формула Erlang B використовується для обчислення ймовірності блокування
// в системі втрат із channels паралельними каналами обслуговування та без
// черги очікування. Параметр traffic задає інтенсивність навантаження в
// Ерлангах, а результатом є ймовірність відмови новому запиту в обслуговуванні.
fn erlang_b(traffic: f64, channels: usize) -> f64 {
    if channels == 0 {
        return 1.0;
    }

    let mut probability = 1.0;
    for m in 1..=channels {
        probability = (traffic * probability) / (m as f64 + traffic * probability);
    }
    probability
}

// Цільова ймовірність блокування.
// P_target = 1 - SLA. Наприклад, для SLA = 99.9% допустима ймовірність
// блокування становить 0.001.
fn target_blocking_probability(sla: f64) -> f64 {
    1.0 - sla / 100.0
}

fn ceil_div(lhs: usize, rhs: usize) -> usize {
    if lhs == 0 { 0 } else { 1 + (lhs - 1) / rhs }
}
```

ДОДАТОК 2. КЛЮЧОВІ ЕЛЕМЕНТИ МОДУЛЯ SCALING БАЛАНСУВАЛЬНИКА (ILB)

```
pub fn spawn_event_loop(
    scaling: Arc<ScalingState>,
    mut event_rx: UnboundedReceiver<ScalingMessage>,
    cluster: Arc<ClusterState>,
) {
    // Запускає асинхронний цикл обробки подій масштабування.
    tokio::spawn(async move {
        while let Some(event) = event_rx.recv().await {
            match event {
                // Обробляє подію про те, що бекенд був повністю злитий (drained)
                // і може бути повернутий до пулу доступних бекендів.
                ScalingMessage::BackendDrained { backend } => {
                    cluster.recycle_drained_backend(&backend);
                    log_info(format!(
                        "scaling event: backend drained and returned to available pool:
 {} available_backends={}",
                        backend,
                        cluster.available_backend_count()
                    ));
                }
                // Обробляє команду на зміну кількості серверів (додати або видалити).
                ScalingMessage::ChangeServers { delta } => {
                    // Якщо delta > 0, то потрібно додати нові сервери
                    if delta > 0 {
                        for _ in 0..delta {
                            let Some(address) = cluster.reserve_next_available_backend
() else {
                                scaling.mark_servers_ready(1);
                                log_error(format!(
                                    "scaling add skipped: no available backend in pool;
 pending_servers={}",
                                    scaling.pending_servers()
```

```
                    ));
                    continue;
                };

                let scaling = Arc::clone(&scaling);
                let cluster = Arc::clone(&cluster);
                // Запускає асинхронне завдання для додавання нового бекенду до кластера.
                tokio::spawn(async move {
                    let backend_start_delay_sec = scaling.backend_start_delay_sec();
                    log_info(format!(
                        "scaling add scheduled: backend={} startup_delay_sec={}",
                        address, backend_start_delay_sec
                    ));
                    // Час необхідний для запуску нового бекенду перед тим,
                    // як він буде доданий до кластера.
                    if backend_start_delay_sec > 0 {
                        time::sleep(Duration::from_secs(backend_start_delay_sec)).await;
                    }

                    match cluster.add_backend(address.as_ref()).await {
                        Ok(backend) => {
                            scaling.mark_servers_ready(1);
                            log_info(format!(
                                "scaling add completed: backend={} active_backends={} available_backends={}",
                                backend,
                                cluster.backend_count(),
                                cluster.available_backend_count()
                            ));
                        }
```

```
                                Err(err) => {
                                    cluster.return_reserved_backend(Arc::clone(&add
ress));
                                    scaling.mark_servers_ready(1);
                                    log_error(format!(
                                        "scaling add failed: backend={} error={} ac
tive_backends={} available_backends={}",
                                        address,
                                        err,
                                        cluster.backend_count(),
                                        cluster.available_backend_count()
                                    ));
                                }
                            }
                        });
                    }
                    // Якщо delta < 0, то потрібно видалити існуючі сервери
                } else if delta < 0 {
                    for _ in 0..(-delta) {
                        match cluster.begin_drain_min_active_backend() {
                            Ok(backend) => {
                                log_info(format!(
                                    "scaling drain started: backend={} active_backe
nds={}",
                                    backend,
                                    cluster.backend_count()
                                ));
                            }
                            Err(err) => {
                                log_error(format!("scaling drain skipped: {}", er
r));
                                break;
                            }
                        }
                    }
```

```
                    }

                    log_info(format!(
                        "scaling command: delta={} pending_servers={} active_backends=
{} available_backends={}",
                        delta,
                        scaling.pending_servers(),
                        cluster.backend_count(),
                        cluster.available_backend_count()
                    ));
                }
            }
        }
    });
}
```

ДОДАТОК 3. КЛЮЧОВІ ЕЛЕМЕНТИ МОДУЛЯ CLUSTER БАЛАНСУВАЛЬНИКА (ILB)

```rust
// Структура для зберігання інформації про бекенд, включаючи його адресу,
// кількість активних запитів, RPS та стан (активний, дренуючий, вимкнений).
#[derive(Debug)]
struct BackendEntry {
    address: Arc<str>,
    active_requests: AtomicUsize,
    request_counter_for_rps: AtomicUsize,
    current_rps: AtomicUsize,
    max_rps: AtomicUsize,
    state: AtomicU8,
}


    // Функція емулю створення нового бекенд сервера. В продуктовій версії має бути
    // замінена на звернення до модуля адаптера хваттного середовища для створення реал
ьного інстансу сервера.
    pub fn acquire_backend(self: &Arc<Self>) -> Result<BackendLease, AcquireBackendErro
r> {
        let backends = self.backends.load();
        if backends.is_empty() {
            return Err(AcquireBackendError::NoRoutableBackends);
        }
        let start_idx = self.rr_counter.fetch_add(1, Ordering::Relaxed);

        for offset in 0..backends.len() {
            let backend_idx = (start_idx + offset) % backends.len();
            let backend = Arc::clone(&backends[backend_idx]);

            if backend.try_acquire_request() {
                return Ok(BackendLease {
                    cluster: Arc::clone(self),
                    backend,
```

```
                });
            }
        }

        Err(AcquireBackendError::NoRoutableBackends)
    }

    // Функція для додавання нового бекенд сервера до кластера. Вона приймає адресу нов
ого сервера,
    // нормалізує її, виконує healthcheck, і якщо все успішно, додає його до пулу бекен
дів.
    // У продуктовій версії має бути замінена на звернення до модуля адаптера хмарного
середовища.
    pub async fn add_backend(&self, address: &str) -> Result<Arc<str>, AddBackendError>
 {
        let normalized = Arc::<str>::from(
            normalize_backend_address(address)
                .map_err(|_| AddBackendError::InvalidAddress(address.trim().to_string
()))?,
        );

        if self
            .backends
            .load()
            .iter()
            .any(|backend| backend.address.as_ref() == normalized.as_ref())
        {
            return Err(AddBackendError::AlreadyExists(normalized.to_string()));
        }

        let client: Client<HttpConnector, Empty<Bytes>> =
            Client::builder(TokioExecutor::new()).build_http();
        healthcheck_backend(&client, &normalized).await?;
        let new_backend = Arc::new(BackendEntry::new(Arc::clone(&normalized)));
```

```
        loop {
            let current = self.backends.load_full();
            if current
                .iter()
                .any(|backend| backend.address.as_ref() == normalized.as_ref())
            {
                return Err(AddBackendError::AlreadyExists(normalized.to_string()));
            }

            let mut updated = Vec::with_capacity(current.len() + 1);
            updated.extend(current.iter().cloned());
            updated.push(Arc::clone(&new_backend));

            let swapped = self.backends.compare_and_swap(&current, Arc::new(updated));

            if Arc::ptr_eq(&swapped, &current) {
                self.remove_available_backend(&normalized);
                return Ok(normalized);
            }
        }
    }

    // Функція для початку процесу дренування бекенд сервера з найменшою кількістю акти
вних запитів.
    pub fn begin_drain_min_active_backend(&self) -> Result<Arc<str>, DrainBackendError>
 {
        loop {
            let backends = self.backends.load();
            let candidate = backends
                .iter()
                .filter(|backend| backend.state() == BackendState::Active)
                .min_by_key(|backend| backend.active_requests())
                .cloned()
```

```
                .ok_or(DrainBackendError::NoActiveBackends)?;

            if candidate.begin_draining() {
                let address = Arc::clone(&candidate.address);
                self.try_finish_draining(&candidate);
                return Ok(address);
            }
        }
    }

    // Функція для видалення дренованого бекенд сервера з пулу.
    // У продуктовій версії має бути замінена на звернення до модуля адаптера хмарного
середовища
    // для видалення реального інстансу сервера.
    pub fn recycle_drained_backend(&self, address: &str) {
        loop {
            let current = self.backends.load_full();
            let mut removed = None;
            let updated: Vec<_> = current
                .iter()
                .filter_map(|backend| {
                    if backend.address.as_ref() == address {
                        removed = Some(Arc::clone(&backend.address));
                        None
                    } else {
                        Some(Arc::clone(backend))
                    }
                })
                .collect();

            let Some(removed_address) = removed else {
                return;
            };
```

```
            let swapped = self.backends.compare_and_swap(&current, Arc::new(updated));
            if Arc::ptr_eq(&swapped, &current) {
                self.return_reserved_backend(removed_address);
                return;
            }
        }
    }
```

ДОДАТОК 4. КЛЮЧОВІ ЕЛЕМЕНТИ МОДУЛЯ ACL БАЛАНСУВАЛЬНИКА (ILB)

```
impl AclState {
    pub fn load_from_file(
        path: impl AsRef<Path>,
        lifetime_secs: u64,
    ) -> Result<Arc<Self>, DynError> {
        let raw = fs::read_to_string(path)?;
        let disk: DiskBlacklist = serde_json::from_str(&raw)?;
        let lifetime = Duration::from_secs(lifetime_secs);
        let now = Instant::now();

        let mut blocked_ips = HashMap::with_capacity(disk.ips.len());
        for ip in disk.ips {
            let parsed = parse_ipv4(&ip)?;
            blocked_ips.insert(parsed, now + lifetime);
        }

        Ok(Arc::new(Self {
            lifetime,
            blocked_ips: RwLock::new(blocked_ips),
            total_checks: AtomicU64::new(0),
            total_positive_checks: AtomicU64::new(0),
            total_negative_checks: AtomicU64::new(0),
            total_redis_messages_received: AtomicU64::new(0),
            redis_messages_counter_for_rps: AtomicU64::new(0),
            current_redis_messages_rps: AtomicU64::new(0),
        }))
    }

    pub fn is_ip_blocked(&self, ip: &str) -> bool {
        self.total_checks.fetch_add(1, Ordering::Relaxed);
```

```
    let parsed = match parse_ipv4(ip) {
        Ok(ip) => ip,
        Err(_) => {
            self.total_negative_checks.fetch_add(1, Ordering::Relaxed);
            return false;
        }
    };

    let now = Instant::now();
    let blocked_ips = self.blocked_ips.read().unwrap();
    let is_blocked = blocked_ips
        .get(&parsed)
        .is_some_and(|expires_at| *expires_at > now);
    if is_blocked {
        self.total_positive_checks.fetch_add(1, Ordering::Relaxed);
    } else {
        self.total_negative_checks.fetch_add(1, Ordering::Relaxed);
    }
    is_blocked
}

pub fn add_ip(
    &self,
    ip: &str,
    lifetime_secs: Option<u64>,
) -> Result<(), AclError> {
    let parsed = parse_ipv4(ip)?;
    let lifetime = lifetime_secs
        .map(Duration::from_secs)
        .unwrap_or(self.lifetime);
    let expires_at = Instant::now() + lifetime;

    let mut blocked_ips = self.blocked_ips.write().unwrap();
    blocked_ips.insert(parsed, expires_at);
```

```
        Ok(())
    }

    pub fn blocked_count(&self) -> usize {
        self.blocked_ips.read().unwrap().len()
    }

    pub fn lifetime_secs(&self) -> u64 {
        self.lifetime.as_secs()
    }

    pub fn sample_metrics(&self) -> AclMetricsSnapshot {
        let redis_messages_rps = self
            .redis_messages_counter_for_rps
            .swap(0, Ordering::AcqRel);
        self.current_redis_messages_rps
            .store(redis_messages_rps, Ordering::Release);

        AclMetricsSnapshot {
            total_checks: self.total_checks.load(Ordering::Acquire),
            total_positive_checks: self.total_positive_checks.load(Ordering::Acquire),
            total_negative_checks: self.total_negative_checks.load(Ordering::Acquire),
            current_list_size: self.blocked_count(),
            total_redis_messages_received: self
                .total_redis_messages_received
                .load(Ordering::Acquire),
            current_redis_messages_rps: self.current_redis_messages_rps.load(Ordering::
Acquire),
        }
    }

    fn record_redis_message(&self) {
        self.total_redis_messages_received
            .fetch_add(1, Ordering::Relaxed);
```

```
            self.redis_messages_counter_for_rps
                .fetch_add(1, Ordering::Relaxed);
        }

        fn cleanup_expired(&self) {
            let now = Instant::now();
            let mut blocked_ips = self.blocked_ips.write().unwrap();
            blocked_ips.retain(|_, expires_at| *expires_at > now);
        }
    }
```

ДОДАТОК 5. СКРИПТ ПОБУДОВИ СЕРТИФІКАТІВ

```
#!/usr/bin/env bash
set -euo pipefail
umask 077

OUT_DIR="${OUT_DIR:-../runtime/pki}"
NUM_SITES="${NUM_SITES:-10000}"
DOMAIN_SUFFIX="${DOMAIN_SUFFIX:-example}"
DAYS_ROOT="${DAYS_ROOT:-5475}"
DAYS_INTERMEDIATE="${DAYS_INTERMEDIATE:-3650}"
DAYS_LEAF="${DAYS_LEAF:-397}"
PROGRESS_EVERY="${PROGRESS_EVERY:-100}"
CA_PUBLIC_DIR="$OUT_DIR/ca/public"
CA_PRIVATE_DIR="$OUT_DIR/ca/private"
DOMAINS_DIR="$OUT_DIR/domains"

mkdir -p "$CA_PUBLIC_DIR" "$CA_PRIVATE_DIR" "$DOMAINS_DIR"

WORK_DIR="$(mktemp -d)"
trap 'rm -rf "$WORK_DIR"' EXIT

echo "[1/4] Генерація Root CA..."

cat > "$WORK_DIR/root.req.cnf" <<'EOF'
[ req ]
prompt = no
utf8 = yes
string_mask = utf8only
distinguished_name = dn
x509_extensions = v3_root_ca
default_md = sha384

[ dn ]
```

```
C = UA
ST = м. Київ
L = м. Київ
O = Олександр Чижов
OU = Root Certification Authority
CN = Олександр Чижов Root CA
emailAddress = oleksandr.chyzhov@npp.kai.ua
streetAddress = просп. Любомира Гузара, 1
postalCode = 03058

[ v3_root_ca ]
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid:always,issuer
basicConstraints = critical, CA:true, pathlen:1
keyUsage = critical, keyCertSign, cRLSign
EOF

openssl ecparam -name secp384r1 -genkey -noout \
  -out "$CA_PRIVATE_DIR/root.key.pem"
openssl req -new -x509 \
  -config "$WORK_DIR/root.req.cnf" \
  -key "$CA_PRIVATE_DIR/root.key.pem" \
  -sha384 \
  -days "$DAYS_ROOT" \
  -set_serial "0x$(openssl rand -hex 16)" \
  -out "$CA_PUBLIC_DIR/root.cert.pem" \
  >/dev/null 2>&1
echo "[2/4] Генерація Intermediate CA..."
cat > "$WORK_DIR/intermediate.req.cnf" <<'EOF'
[ req ]
prompt = no
utf8 = yes
string_mask = utf8only
distinguished_name = dn
```

```
default_md = sha384

[ dn ]
C = UA
ST = м. Київ
L = м. Київ
O = Олександр Чижов
OU = Issuing Certification Authority
CN = Олександр Чижов Intermediate CA
emailAddress = oleksandr.chyzhov@npp.kai.ua
streetAddress = просп. Любомира Гузара, 1
postalCode = 03058
EOF

cat > "$WORK_DIR/intermediate.ext.cnf" <<'EOF'
[ v3_intermediate_ca ]
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid,issuer
basicConstraints = critical, CA:true, pathlen:0
keyUsage = critical, keyCertSign, cRLSign
EOF

openssl ecparam -name secp384r1 -genkey -noout \
  -out "$CA_PRIVATE_DIR/intermediate.key.pem"
openssl req -new \
  -config "$WORK_DIR/intermediate.req.cnf" \
  -key "$CA_PRIVATE_DIR/intermediate.key.pem" \
  -out "$WORK_DIR/intermediate.csr.pem" \
  >/dev/null 2>&1
openssl x509 -req \
  -in "$WORK_DIR/intermediate.csr.pem" \
  -CA "$CA_PUBLIC_DIR/root.cert.pem" \
  -CAkey "$CA_PRIVATE_DIR/root.key.pem" \
  -sha384 \
```

```
  -days "$DAYS_INTERMEDIATE" \
  -set_serial "0x$(openssl rand -hex 16)" \
  -extfile "$WORK_DIR/intermediate.ext.cnf" \
  -extensions v3_intermediate_ca \
  -out "$CA_PUBLIC_DIR/intermediate.cert.pem" \
  >/dev/null 2>&1

echo "[3/4] Генерація $NUM_SITES сертифікатів..."

for i in $(seq 1 "$NUM_SITES"); do
  idx="$(printf '%06d' "$i")"
  fqdn="site${idx}.${DOMAIN_SUFFIX}"
  site_dir="$DOMAINS_DIR/$fqdn"

  mkdir -p "$site_dir"

  site_email="oleksandr${idx}@npp.kai.ua"
  site_given_name="Олександр${idx}"
  site_street="м. Київ ${idx}"

  leaf_req="$WORK_DIR/${fqdn}.req.cnf"
  leaf_ext="$WORK_DIR/${fqdn}.ext.cnf"
  leaf_csr="$WORK_DIR/${fqdn}.csr.pem"
  leaf_only_cert="$WORK_DIR/${fqdn}.leaf.cert.pem"

  cat > "$leaf_req" <<EOF
[ req ]
prompt = no
utf8 = yes
string_mask = utf8only
distinguished_name = dn
req_extensions = req_ext
default_md = sha256
```

```
[ dn ]
C = UA
ST = м. Київ
L = м. Київ
O = Test Infrastructure
OU = ILB Experimental Stand
CN = ${fqdn}
emailAddress = ${site_email}
streetAddress = ${site_street}
postalCode = 03058
givenName = ${site_given_name}
surname = Чижов
serialNumber = SITE${idx}

[ req_ext ]
subjectAltName = @alt_names

[ alt_names ]
DNS.1 = ${fqdn}
EOF

  cat > "$leaf_ext" <<EOF
[ v3_leaf ]
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid,issuer
basicConstraints = critical, CA:false
keyUsage = critical, digitalSignature
extendedKeyUsage = serverAuth
subjectAltName = @alt_names

[ alt_names ]
DNS.1 = ${fqdn}
EOF
```

```
  openssl ecparam -name prime256v1 -genkey -noout \
    -out "$site_dir/key.pem"
  openssl req -new \
    -config "$leaf_req" \
    -key "$site_dir/key.pem" \
    -out "$leaf_csr" \
    >/dev/null 2>&1
  serial_hex="$(printf '03%030x' "$i")"
  openssl x509 -req \
    -in "$leaf_csr" \
    -CA "$CA_PUBLIC_DIR/intermediate.cert.pem" \
    -CAkey "$CA_PRIVATE_DIR/intermediate.key.pem" \
    -sha256 \
    -days "$DAYS_LEAF" \
    -set_serial "0x${serial_hex}" \
    -extfile "$leaf_ext" \
    -extensions v3_leaf \
    -out "$leaf_only_cert" \
    >/dev/null 2>&1

  cat "$leaf_only_cert" "$CA_PUBLIC_DIR/intermediate.cert.pem" > "$site_dir/cert.pem"

  rm -f "$leaf_req" "$leaf_ext" "$leaf_csr" "$leaf_only_cert"

  if (( i % PROGRESS_EVERY == 0 || i == NUM_SITES )); then
    echo "  issued: $i / $NUM_SITES"
  fi
done

echo "[4/4] Завершення..."

cat > "$OUT_DIR/README.txt" <<EOF
```

```
Структура вихідних файлів:

$CA_PUBLIC_DIR/root.cert.pem
  Кореневий сертифікат, який треба довірити клієнтам/браузерам тестового стенду.

$CA_PUBLIC_DIR/intermediate.cert.pem
  Сертифікат проміжного центру сертифікації.

$CA_PRIVATE_DIR/root.key.pem
$CA_PRIVATE_DIR/intermediate.key.pem
  Приватні ключі Root та Intermediate CA.

$DOMAINS_DIR/<site-fqdn>/key.pem
  Приватний ключ сертифіката сайту.

$DOMAINS_DIR/<site-fqdn>/cert.pem
  Готовий PEM для сервера: leaf + intermediate.
  Це файл, який можна одразу віддавати балансувальнику як сертифікат домену.

Root у cert.pem навмисно не додається.
EOF

find "$OUT_DIR" -type f -name "*key.pem" -exec chmod 600 {} \;
find "$OUT_DIR" -type f \( -name "*.cert.pem" -o -name "cert.pem" \) -exec chmod 644 {}
 \;
chmod 644 "$OUT_DIR/README.txt"

echo
echo "Готово."
echo "Каталог виходу: $OUT_DIR"
echo "Root cert       : $CA_PUBLIC_DIR/root.cert.pem"
echo "Intermediate    : $CA_PUBLIC_DIR/intermediate.cert.pem"
echo "Domains         : $DOMAINS_DIR"
```

```
echo
echo "Приклад готового домену:"
echo "  $DOMAINS_DIR/site000001.${DOMAIN_SUFFIX}/key.pem"
echo "  $DOMAINS_DIR/site000001.${DOMAIN_SUFFIX}/cert.pem"
```

ДОДАТОК 6. СКРИПТ K6 ЕМУЛЯТОРА НАВАНТАЖЕННЯ ДЛЯ ТЕСТУВАННЯ BACKEND СЕРВЕРІВ

```
// Прямий RPS-тест швидкості бекенду.
//
// Цей сценарій відправляє трафік безпосередньо на фіксований набір портів бекенду, без
 ILB попереду.
// Загальна кількість RPS зростає від START_RPS до END_RPS різкими кроками кожні STEP_D
URATION_SECONDS.
// Трафік розподіляється рівномірно між SERVER_COUNT цілями шляхом запуску одного сцена
рію k6 на кожне запущення бекенду.
// Підсумок показує як поведінку на кожному кроці, так і загальні результати на цільове
 сховище, щоб було легко бачити,
// при якому загальному навантаженню першими з'являються збої або втрати пропускної зда
тності.


import exec from 'k6/execution';
import http from 'k6/http';


const HOST = __ENV.HOST || '10.0.0.3';
const START_PORT = Number(__ENV.START_PORT || 18081);
const SERVER_COUNT = Number(__ENV.SERVER_COUNT || 10);
const START_RPS = Number(__ENV.START_RPS || 1000);
const END_RPS = Number(__ENV.END_RPS || 5000);
const RPS_STEP = Number(__ENV.RPS_STEP || 100);
const STEP_DURATION_SECONDS = Number(__ENV.STEP_DURATION_SECONDS || 5);
const REQUEST_TIMEOUT = __ENV.REQUEST_TIMEOUT || '15s';
const TOTAL_PRE_ALLOCATED_VUS = Number(__ENV.PRE_ALLOCATED_VUS || 6000);
const TOTAL_MAX_VUS = Number(__ENV.MAX_VUS || 15000);
const SUMMARY_TXT = __ENV.SUMMARY_TXT || 'backend-rps-summary.txt';
const SUMMARY_JSON = __ENV.SUMMARY_JSON || 'backend-rps-summary.json';


if (START_RPS % SERVER_COUNT !== 0) {
  throw new Error(`START_RPS=${START_RPS} must be divisible by SERVER_COUNT=${SERVER_CO
```

```
UNT}`);
}

if (END_RPS % SERVER_COUNT !== 0) {
  throw new Error(`END_RPS=${END_RPS} must be divisible by SERVER_COUNT=${SERVER_COUNT}
`);
}

if (RPS_STEP % SERVER_COUNT !== 0) {
  throw new Error(`RPS_STEP=${RPS_STEP} must be divisible by SERVER_COUNT=${SERVER_COUN
T}`);
}

const TARGETS = Array.from({ length: SERVER_COUNT }, (_, index) => ({
  id: `backend_${START_PORT + index}`,
  url: `http://${HOST}:${START_PORT + index}`,
  port: START_PORT + index,
}));

const LOAD_STEPS = [];
for (let rps = START_RPS; rps <= END_RPS; rps += RPS_STEP) {
  LOAD_STEPS.push(rps);
}

const PER_SERVER_START_RPS = START_RPS / SERVER_COUNT;
const PER_SERVER_END_RPS = END_RPS / SERVER_COUNT;
const PER_SERVER_RPS_STEP = RPS_STEP / SERVER_COUNT;
const PRE_ALLOCATED_VUS_PER_SERVER = Math.ceil(TOTAL_PRE_ALLOCATED_VUS / SERVER_COUNT);
const MAX_VUS_PER_SERVER = Math.ceil(TOTAL_MAX_VUS / SERVER_COUNT);

const PER_SERVER_STAGES = [];
for (
  let perServerRps = PER_SERVER_START_RPS;
```

```
  perServerRps <= PER_SERVER_END_RPS;
  perServerRps += PER_SERVER_RPS_STEP
) {
  const holdStage = { target: perServerRps, duration: `${STEP_DURATION_SECONDS}s` };
  if (perServerRps === PER_SERVER_START_RPS) {
    PER_SERVER_STAGES.push(holdStage);
  } else {
    PER_SERVER_STAGES.push({ target: perServerRps, duration: '0s' });
    PER_SERVER_STAGES.push(holdStage);
  }
}

const thresholds = {};
for (const rps of LOAD_STEPS) {
  const selector = `load_band:rps_${rps}`;
  thresholds[`http_reqs{${selector}}`] = ['count>=0'];
  thresholds[`http_req_failed{${selector}}`] = ['rate>=0'];
  thresholds[`http_req_duration{${selector}}`] = ['avg>=0'];
}

for (const target of TARGETS) {
  const selector = `target_port:${target.port}`;
  thresholds[`http_reqs{${selector}}`] = ['count>=0'];
  thresholds[`http_req_failed{${selector}}`] = ['rate>=0'];
  thresholds[`http_req_duration{${selector}}`] = ['avg>=0'];
}

const scenarios = {};
for (const target of TARGETS) {
  scenarios[target.id] = {
    executor: 'ramping-arrival-rate',
    exec: 'hitBackend',
    startRate: PER_SERVER_START_RPS,
    timeUnit: '1s',
```

```
    preAllocatedVUs: PRE_ALLOCATED_VUS_PER_SERVER,
    maxVUs: MAX_VUS_PER_SERVER,
    stages: PER_SERVER_STAGES,
    gracefulStop: '5s',
    tags: {
      target: 'backend',
      target_port: String(target.port),
      target_url: target.url,
      test_case: 'backend_rps_steps',
    },
  };
}

export const options = {
  discardResponseBodies: true,
  summaryTrendStats: ['avg', 'min', 'med', 'p(90)', 'p(95)', 'max'],
  thresholds,
  scenarios,
};

function currentRpsLevel() {
  const elapsedMs = exec.instance.currentTestRunDuration;
  const stepIndex = Math.min(
    Math.floor(elapsedMs / (STEP_DURATION_SECONDS * 1000)),
    LOAD_STEPS.length - 1,
  );

  return LOAD_STEPS[stepIndex];
}

function currentTarget() {
  const scenarioName = exec.scenario.name;
  const target = TARGETS.find((item) => item.id === scenarioName);
  if (!target) {
```

```
    throw new Error(`Unknown scenario name: ${scenarioName}`);
  }

  return target;
}

function metricValues(data, metricName, selector) {
  const metric = data.metrics[`${metricName}{${selector}}`];
  return metric ? metric.values : null;
}

function summaryStatus(expectedRequests, actualRequests, failRate) {
  const shortfallRatio = expectedRequests === 0
    ? 0
    : (expectedRequests - actualRequests) / expectedRequests;

  if (failRate > 0 || shortfallRatio > 0.02) {
    return 'ISSUES';
  }

  return 'OK';
}

function buildSummary(data) {
  const lines = [];
  const perStep = [];
  const perTarget = [];

  lines.push('Backend RPS step test');
  lines.push(`Targets: ${TARGETS.map((target) => target.url).join(', ')}`);
  lines.push(
    `Profile: ${START_RPS}->${END_RPS} total RPS, +${RPS_STEP} every ${STEP_DURATION_SE
CONDS}s`,
```

```
  );
  lines.push(
    `Per target: ${PER_SERVER_START_RPS}->${PER_SERVER_END_RPS} RPS, +${PER_SERVER_RPS_
STEP} every ${STEP_DURATION_SECONDS}s`,
  );
  lines.push(
    `VUs total: preAllocated=${TOTAL_PRE_ALLOCATED_VUS}, max=${TOTAL_MAX_VUS}`,
  );
  lines.push(
    `VUs per target: preAllocated=${PRE_ALLOCATED_VUS_PER_SERVER}, max=${MAX_VUS_PER_SE
RVER}, timeout=${REQUEST_TIMEOUT}`,
  );
  lines.push('');
  lines.push('Per step');
  lines.push('RPS | Expected | Sent | Loss% | Fail% | Avg ms | P95 ms | Status');

  for (const rps of LOAD_STEPS) {
    const selector = `load_band:rps_${rps}`;
    const requestValues = metricValues(data, 'http_reqs', selector) || {};
    const failedValues = metricValues(data, 'http_req_failed', selector) || {};
    const durationValues = metricValues(data, 'http_req_duration', selector) || {};

    const expectedRequests = rps * STEP_DURATION_SECONDS;
    const actualRequests = Number(requestValues.count || 0);
    const failRate = Number(failedValues.rate || 0);
    const avgMs = Number(durationValues.avg || 0);
    const p95Ms = Number(durationValues['p(95)'] || 0);
    const lossPercent = expectedRequests === 0
      ? 0
      : Math.max(0, ((expectedRequests - actualRequests) / expectedRequests) * 100);
    const status = summaryStatus(expectedRequests, actualRequests, failRate);

    perStep.push({
      rps,
```

```
    expectedRequests,
    actualRequests,
    lossPercent,
    failRate,
    avgMs,
    p95Ms,
    status,
  });

  lines.push(
    [
      String(rps).padStart(4, ' '),
      String(expectedRequests).padStart(8, ' '),
      String(actualRequests).padStart(5, ' '),
      `${lossPercent.toFixed(2)}%`.padStart(6, ' '),
      `${(failRate * 100).toFixed(2)}%`.padStart(6, ' '),
      avgMs.toFixed(2).padStart(7, ' '),
      p95Ms.toFixed(2).padStart(7, ' '),
      status.padStart(6, ' '),
    ].join(' | '),
  );
}

lines.push('');
lines.push('Per target');
lines.push('Port  | Requests | Fail% | Avg ms | P95 ms');

for (const target of TARGETS) {
  const selector = `target_port:${target.port}`;
  const requestValues = metricValues(data, 'http_reqs', selector) || {};
  const failedValues = metricValues(data, 'http_req_failed', selector) || {};
  const durationValues = metricValues(data, 'http_req_duration', selector) || {};

  const requests = Number(requestValues.count || 0);
```

```
  const failRate = Number(failedValues.rate || 0);
  const avgMs = Number(durationValues.avg || 0);
  const p95Ms = Number(durationValues['p(95)'] || 0);

  perTarget.push({
    port: target.port,
    url: target.url,
    requests,
    failRate,
    avgMs,
    p95Ms,
  });

  lines.push(
    [
      String(target.port).padStart(5, ' '),
      String(requests).padStart(8, ' '),
      `${(failRate * 100).toFixed(2)}%`.padStart(6, ' '),
      avgMs.toFixed(2).padStart(7, ' '),
      p95Ms.toFixed(2).padStart(7, ' '),
    ].join(' | '),
  );
}

const summaryJson = {
  generatedAt: new Date().toISOString(),
  config: {
    host: HOST,
    startPort: START_PORT,
    serverCount: SERVER_COUNT,
    targets: TARGETS,
    startRps: START_RPS,
    endRps: END_RPS,
    rpsStep: RPS_STEP,
```

```
      stepDurationSeconds: STEP_DURATION_SECONDS,
      requestTimeout: REQUEST_TIMEOUT,
      totalPreAllocatedVUs: TOTAL_PRE_ALLOCATED_VUS,
      totalMaxVUs: TOTAL_MAX_VUS,
      perTargetPreAllocatedVUs: PRE_ALLOCATED_VUS_PER_SERVER,
      perTargetMaxVUs: MAX_VUS_PER_SERVER,
      perServerStartRps: PER_SERVER_START_RPS,
      perServerEndRps: PER_SERVER_END_RPS,
      perServerRpsStep: PER_SERVER_RPS_STEP,
    },
    overall: {
      httpReqs: data.metrics.http_reqs ? data.metrics.http_reqs.values : null,
      httpReqFailed: data.metrics.http_req_failed
        ? data.metrics.http_req_failed.values
        : null,
      httpReqDuration: data.metrics.http_req_duration
        ? data.metrics.http_req_duration.values
        : null,
      droppedIterations: data.metrics.dropped_iterations
        ? data.metrics.dropped_iterations.values
        : null,
    },
    perStep,
    perTarget,
  };

  return {
    text: `${lines.join('\n')}\n`,
    json: JSON.stringify(summaryJson, null, 2),
  };
}

export function hitBackend() {
  const target = currentTarget();
```

```
  const currentRps = currentRpsLevel();

  http.get(target.url, {
    timeout: REQUEST_TIMEOUT,
    tags: {
      load_band: `rps_${currentRps}`,
    },
  });
}

export function handleSummary(data) {
  const summary = buildSummary(data);

  return {
    stdout: summary.text,
    [SUMMARY_TXT]: summary.text,
    [SUMMARY_JSON]: summary.json,
  };
}
```

ДОДАТОК 7. ТИЖНЕВИЙ НОРМАЛІЗОВАНИЙ ПРОФІЛЬ НАВАНТАЖЕННЯ

Наведено перші декілька рядків профілю навантаження. Повний профіль налічує 5040 рядків.

Хвилина	День тижня	Час	RPS
0	Пн	00:00	45.501
2	Пн	00:02	44.61
4	Пн	00:04	44.188
6	Пн	00:06	43.393
8	Пн	00:08	41.831
10	Пн	00:10	44.611
12	Пн	00:12	45.063
14	Пн	00:14	42.336
16	Пн	00:16	42.638
18	Пн	00:18	41.639
20	Пн	00:20	38.963
22	Пн	00:22	41.62
24	Пн	00:24	39.11
26	Пн	00:26	40.339
28	Пн	00:28	35.423
30	Пн	00:30	40.178
32	Пн	00:32	34.593
34	Пн	00:34	37.521
36	Пн	00:36	32.942
38	Пн	00:38	35.452
40	Пн	00:40	35.72
42	Пн	00:42	35.517
44	Пн	00:44	35.968
46	Пн	00:46	38.369
48	Пн	00:48	34.405
50	Пн	00:50	35.828
52	Пн	00:52	32.157
54	Пн	00:54	31.332
56	Пн	00:56	30.648

ДОДАТОК 8. ДОКУМЕНТИ ВПРОВАДЖЕННЯ ОСНОВНИХ РЕЗУЛЬТАТІВ ДИСЕРТАЦІЙНОЇ РОБОТИ

Товариство з обмеженою відповідальністю
«Акссонсофт»

04119, Україна, м. Київ, вул. Дегтярівська, буд. 25-А, корп. 1, оф. 414
Код ЄДРПОУ- 38507216,
IBAN UA193348510000000026002260745
в АТ "ПУМБ" м. Київ, МФО 334851
тел.+38044 333-71-90, моб. +38 068 333 71 90
https://ua.axxonsoft.com/

АКТ

впровадження результатів дисертаційного дослідження
здобувача ступеня доктора філософії за спеціальністю 122 «Комп'ютерні науки»
ЧИЖОВА Олександра Вікторовича
на тему «Інформаційна технологія оптимізації використання обчислювальних ресурсів у хмарних кластерах для динамічного хостингу веб-сервісів»

Комісія у складі директора Курінного Олександра Васильовича, розробника комп'ютерних програм Посохова Валерія Володимировича та інженера з налагодження й випробувань Бойчука Владислава Олеговича підтверджує, що результати дисертаційної роботи ЧИЖОВА Олександра Вікторовича використовуються в компанії ТОВ «Акссонсофт», а саме:

Запропонована автором математична модель (на базі модифікованої формули Ерланга) використана для аудиту поточної ефективності використання серверного парку компанії. Розрахунки підтвердили можливість скорочення витрат на інфраструктуру на 15-20% при переході до кластерної архітектури.

На тестовому сегменті мережі компанії проведено випробування прототипу балансувальника навантаження з модулем асинхронної фільтрації. Підтверджено здатність алгоритму виявляти аномальний трафік без внесення додаткових затримок (Latency) в обробку легітимних запитів.

Рекомендації автора щодо міграції веб-ресурсів у хмарне середовище (Cloud Migration Framework) прийняті технічним департаментом для використання в процесі модернізації та підвищення ефективності на 2026-2027 роки.

Висновок: Результати дисертації оцінюються як такі, що мають практичну цінність та дозволяють підвищити конкурентоспроможність послуг компанії.

Посохов В.В. розробник комп'ютерних програм
Бойчук В.О. інженер з налагодження й випробувань

З повагою,
Директор **О.В. Курінний**

«ЗАТВЕРДЖУЮ»
Проректор з наукових досліджень та
трансферу технологій
Державного університету
«Київський авіаційний інститут»
_____ Сергій ГНАТЮК
«30» 01 2026 р.

АКТ

про впровадження результатів дисертаційного дослідження

Комісія у складі:

голови – завідувача кафедри комп'ютерних інформаційних технологій, д.т.н., проф. САВЧЕНКО Аліна Станіславівна.

членів комісії: – доцента кафедри комп'ютерних інформаційних технологій, к.т.н., доц. СИДОРЕНКО Вікторія Миколаївна.

– заступника декана Факультету комп'ютерних наук та технологій, к.т.н., ст. дослідник ОХРІМЕНКО Тетяна Олександрівна.

цим Актом засвідчує, що результати дисертаційного дослідження аспіранта кафедри комп'ютерних інформаційних технологій ЧИЖОВА Олександра Вікторовича на тему «Інформаційна технологія оптимізації використання обчислювальних ресурсів у хмарних кластерах для динамічного хостингу веб-сервісів» на здобуття ступеня доктора філософії за спеціальністю 122 «Комп'ютерні науки» впроваджені у навчальний процес та використовуються на кафедрі комп'ютерних інформаційних технологій з 2024-2026 р. при викладанні навчальних дисциплін «Хмарні платформи та технології» та «Паралельні та розподільні обчислення» на кафедри у навчальний процес Державного університету «Київський авіаційний інститут».

Зокрема, у процесі підготовки та викладання лекційних курсів, проведення практичних і лабораторних занять використано аналітичні матеріали щодо динамічного розподілу обчислювального ресурсу в хмарних середовищах та асинхронних методів обробки даних. Результатом впровадження стало формування у здобувачів знань про переваги хмарних середовищ за умови ефективного використання ресурсів, практичне засвоєння підходів до динамічного розподілу ресурсів з метою оптимізації їх використання та розвиток навичок застосування алгоритмів асинхронної обробки даних у процесах, критичних до швидкості опрацювання.

Голова комісії, завідувач кафедри
комп'ютерних інформаційних технологій Аліна САВЧЕНКО

Члени комісії:
доцент кафедри комп'ютерних
інформаційних технологій Вікторія СИДОРЕНКО

Заступник декана факультету комп'ютерних наук та
технологій Тетяна ОХРІМЕНКО