Міністерство освіти і науки України
Чернівецький національний університет імені Юрія Федьковича

Кваліфікаційна наукова праця
на правах рукопису

КИРИЧЕНКО ОЛЕКСАНДР ОЛЕКСІЙОВИЧ

УДК 004.75+004.94]:519.87

ДИСЕРТАЦІЯ

ОПТИМІЗАЦІЯ БЕЗСЕРВЕРНИХ ОБЧИСЛЕНЬ У ХМАРНИХ СЕРЕДОВИЩАХ

121 – інженерія програмного забезпечення

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.О. Кириченко

Науковий керівник: **Остапов Сергій Едуардович** доктор фізико-математичних наук, професор

Чернівці – 2026

АНОТАЦІЯ

Кириченко О.О. **Оптимізація безсерверних обчислень у хмарних середовищах.** – Кваліфікаційна наукова праця на правах рукопису.

Дисертація на здобуття наукового ступеня доктора філософії за спеціальністю 121 – «Інженерія програмного забезпечення» – Чернівецький національний університет імені Юрія Федьковича, Чернівці, 2026.

Дисертаційна робота присвячена оптимізації безсерверних обчислень у хмарних середовищах в умовах нерівномірного навантаження та неоднорідності вхідних потоків завдань. Сучасні підходи до автоматичного масштабування обчислювальних ресурсів мають переважно реактивний характер, що призводить до проблем холодного старту, неефективного використання ресурсів та збільшення затримок у системах черг. Застосування методів машинного навчання та нейронних мереж для прогнозування навантаження потребує значних обсягів історичних наборів даних і не забезпечує аналітичних гарантій якості обслуговування. У дисертаційній роботі розв'язується актуальна науково-прикладна задача розробки інформаційної технології оптимізації безсерверних обчислень у хмарних середовищах шляхом побудови математичних моделей на основі теорії систем масового обслуговування неоднорідної структури та створення відповідного програмного забезпечення.

Об’єктом дослідження є процеси розподіленої обробки даних у хмарних середовищах з використанням безсерверних технологій та систем керування чергами повідомлень.

Предметом дослідження є методи та моделі оптимізації безсерверних обчислень у хмарних середовищах на основі теорії систем масового обслуговування неоднорідної структури.

Метою дослідження є розробка підходів оптимізації безсерверних обчислень у хмарних середовищах, що забезпечують підвищення продуктивності, зниження витрат і мінімізацію холодних стартів шляхом побудови інформаційної технології на основі теорії масового обслуговування

для прогнозування навантаження та проактивного масштабування обчислювальних ресурсів.

У дисертаційній роботі **значно удосконалено** математичну модель безсерверної обчислювальної системи на прикладі хмарної платформи AWS Lambda як неоднорідної системи масового обслуговування зі змішаними режимами роботи, у якій вхідний процес визначається сумішшю незалежних неоднорідних пуассонівських процесів з різних джерел подій. **Встановлено необхідну та достатню умову** обмеженості черги для неоднорідної системи масового обслуговування. **Вперше доведено** граничні еволюції для процесу довжини черги у схемі усереднення та схемі дифузійної апроксимації з використанням апарату напівмарковських випадкових еволюцій, що базується на теорії Марковських ланцюгів та напівмарковських процесів. Схема усереднення визначає детерміновану траєкторію середнього навантаження, а схема дифузійної апроксимації описує випадкові відхилення від цієї траєкторії через стохастичне диференціальне рівняння Ito, що є основою для прогнозування поведінки інтелектуальної системи автоматичного масштабування.

У дисертаційній роботі також **удосконалено алгоритм** параметричної оцінки та оптимізації конфігурації безсерверної системи, який, на відміну від існуючих підходів на основі машинного навчання та нейронних мереж, використовує аналітичні оцінки теорії масового обслуговування, дозволяючи отримати явні формули для оптимальних параметрів та уникнути необхідності навчання моделей штучного інтелекту на великих наборах даних.

Розроблено архітектуру фреймворку для розподіленої обробки даних з використанням безсерверних технологій на хмарній платформі AWS. Особливістю даної архітектури є використання системи черг повідомлень, реалізація слабкої зв'язності між окремими елементами інтелектуальної системи, незалежне їх масштабування в разі потреби, швидке відновлення після збоїв та оптимізація використання обчислювальних ресурсів. **Розроблено та реалізовано інформаційну технологію** для розподіленої

обробки даних з використанням безсерверних обчислень та проактивним автоматичним масштабуванням обчислювальних ресурсів, що є основою створеного програмного забезпечення.

Практичне значення отриманих результатів полягає у реалізації інформаційної технології оптимізації безсерверних обчислень та програмного забезпечення, що забезпечує прогнозування навантаження та проактивне масштабування ресурсів на хмарній платформі. Експериментальна апробація на реальних наборах даних продемонструвала, що аналітична модель на основі Марковських ланцюгів та напівмарковських процесів забезпечила прискорення обробки даних на 25,8%, зростання пропускної здатності на 21,3% та зменшення холодних стартів до 3% порівняно з класичним реактивним масштабуванням. Порівняння підходів показало, що аналітична модель перевищує алгоритм на основі машинного навчання за основними показниками продуктивності та забезпечує гарантії якості обслуговування без потреби у великих обсягах історичних наборів даних.

Результати інтелектуального аналізу даних та прогнозування, отримані за допомогою розробленої інформаційної технології, можуть бути використані для оптимізації безсерверних обчислень у хмарних середовищах. Розроблене програмне забезпечення є відкритим фреймворком, що може бути адаптоване для різних хмарних платформ та використане для побудови інтелектуальних систем автоматичного масштабування ресурсів на основі запропонованих моделей та алгоритмів.

Дисертація складається зі вступу, чотирьох розділів, висновків, переліку використаних джерел та чотирьох додатків.

У **вступі** обґрунтовано актуальність теми дослідження, сформульовано мету, предмет, об’єкт, завдання та методи дослідження, вказано наукову новизну, подано та проаналізовано зв’язок роботи з науковими темами. Зазначено особистий внесок здобувача, а також наведено відомості про апробацію та публікації основних результатів дисертації. Описано структуру та обсяг дисертаційної роботи.

Перший розділ дисертації містить теоретичні основи та огляд літератури у галузі безсерверних обчислень на хмарних платформах. Розглянуто ключові концепції парадигми безсерверних обчислень, зокрема моделі Function-as-a-Service та Backend-as-a-Service, архітектурні особливості подійно-орієнтованих систем, механізми автоматичного масштабування обчислювальних ресурсів та інтеграцію з мікросервісними архітектурами. Проаналізовано основні переваги та обмеження безсерверних обчислень, серед яких найбільш критичними визначено проблему холодного старту, залежність від постачальника хмарних послуг та складність моніторингу розподілених систем. Систематизовано метрики оцінювання ефективності безсерверних інформаційних систем та визначено відкриті проблеми, що потребують подальших досліджень та оптимізації.

Другий розділ присвячений аналізу архітектурних рішень для безсерверних обчислень та розробці архітектури фреймворку для розподіленої обробки даних. Проаналізовано роль подієво-орієнтованих архітектур у побудові інтелектуальних систем на хмарних платформах та проведено порівняльний аналіз ключових шаблонів комунікації. Обґрунтовано вибір системи черг повідомлень як базового компонента архітектури для задач асинхронної розподіленої обробки даних, що забезпечує буферизацію та згладжування пікових навантажень. Проведено детальний порівняльний аналіз підходів до реалізації комунікації в реальному часі. Визначено обмеження реактивного масштабування, зокрема холодні старти та непередбачувані затримки, що підтверджує актуальність розробки проактивних алгоритмів масштабування на основі прогнозування навантаження.

У **третьому розділі** побудовано математичну модель безсерверної інформаційної системи як неоднорідної системи масового обслуговування зі змішаними режимами роботи та розроблено методи оцінювання її ключових параметрів. Досліджено властивості моделі суміші потоків завдань, що формується як поєднання незалежних неоднорідних пуассонівських процесів, та встановлено її відмінності від класичного Markov-Modulated Poisson

Process. Доведено необхідну та достатню умову обмеженості системи черг та граничні еволюції процесу довжини черги у схемі усереднення та схемі дифузійної апроксимації з використанням апарату напівмарковських випадкових еволюцій, що базується на теорії Марковських ланцюгів та напівмарковських процесів. Розроблено метод оцінки параметрів суміші вхідного процесу на основі EM-алгоритму з використанням метрик хмарної платформи AWS. Запропоновано алгоритм параметричної оцінки та оптимізації конфігурації безсерверної системи, який на відміну від підходів на основі машинного навчання та нейронних мереж використовує аналітичні оцінки, що дозволяє отримати явний вигляд для прогнозування оптимальних параметрів без потреби у великих наборах даних для навчання моделей штучного інтелекту.

У **четвертому розділі** здійснено практичну верифікацію теоретичних результатів дисертаційного дослідження та реалізовано програмне забезпечення інформаційної технології оптимізації безсерверних обчислень на хмарній платформі AWS. У першому експерименті досліджено прогнозне автомасштабування на основі нейронних мереж DeepAR, яке на реальних наборах даних зменшило кількість холодних стартів на 27% та кількість необроблених запитів на 14%, при цьому встановлено обмеження підходу, зокрема потребу у великих обсягах історичних даних для навчання моделі. У другому експерименті досліджено прогнозне автомасштабування на основі аналітичної моделі з напівмарковськими процесами, що забезпечило прискорення обробки на 25,8% та зменшення холодних стартів до 3%. У третьому експерименті проведено імітаційне моделювання інтелектуальної системи методом Монте-Карло, що підтвердило переваги моделі суміші потоків. Результати інтелектуального аналізу даних показали, що аналітична модель перевищує алгоритм на основі машинного навчання за основними показниками продуктивності та забезпечує гарантії якості обслуговування та підтверджує ефективність розробленої інформаційної технології та програмного забезпечення для оптимізації безсерверних обчислень.

У **висновках** підсумовано основні результати дисертаційного дослідження.

У **додатках** подано наукові публікації, в яких відображено основні наукові результати роботи, відомості про апробацію результатів дисертації – акти та довідки про впровадження результатів роботи, лістинг частини коду програмного забезпечення.

Запропонована інформаційна технологія для прогнозного автомасштабування, що дозволяє заздалегідь розгортати обчислювальні ресурси на хмарній платформі використовується у роботі компанії Finker Finance B.V. та ФОП Вербицької С.І. А результати теоретичних та практичних досліджень використовуються у навчальному процесі кафедри математичних проблем управління і кібернетики та кафедри програмного забезпечення комп'ютерних систем Чернівецького національного університету імені Юрія Федьковича.

Ключові слова: інформаційна технологія, хмарна платформа, оптимізація, модель, алгоритм, машинне навчання, нейронні мережі, штучний інтелект, система черг, інтелектуальна система, інтелектуальний аналіз даних, Марковські та напівмарковські ланцюги, прогнозування, набір даних, програмне забезпечення.

ABSTRACT

Kyrychenko O. **Optimization of serverless computing in cloud environments.** – Qualification research work published in the manuscript.

Dissertation for the degree of Doctor of Philosophy, speciality 121 – "Software Engineering" – Yuriy Fedkovych Chernivtsi National University, Chernivtsi, 2026.

This dissertation addresses the optimization of serverless computing in cloud environments under conditions of uneven workloads and heterogeneous incoming task streams. Existing approaches to automatic scaling of computing resources are predominantly reactive, leading to cold-start problems, inefficient resource

utilization, and increased latency in queueing systems. Although machine learning methods and neural networks can be used for workload prediction, they require large volumes of historical data and do not provide analytical guarantees of quality of service. The dissertation addresses the relevant scientific and applied problem of developing information technology to optimize serverless computing in cloud environments by constructing mathematical models based on the theory of heterogeneous queueing systems and developing the corresponding software.

The object of the study is the processes of distributed data processing in cloud environments using serverless technologies and message queue management systems.

The subject of the study is the methods and models for optimizing serverless computing in cloud environments using the theory of heterogeneous queueing systems.

The study aims to develop approaches to optimizing serverless computing in cloud environments that improve performance, reduce costs, and minimize cold starts by creating an information technology based on queueing theory for workload prediction and proactive scaling of computing resources.

The dissertation has **significantly further developed and improved** the mathematical model of a serverless computing system, using the AWS Lambda cloud platform as an example, by representing it as a heterogeneous queueing system with mixed operating modes, in which the input process is defined as a mixture of independent, non-homogeneous Poisson processes generated by different event sources. **A necessary and sufficient condition for stability condition** for the queue in such a heterogeneous queueing system **is established**. **For the first time**, limiting results for the queue-length process **are obtained** under both the averaging and diffusion approximation schemes, using the framework of semi-Markov random evolutions based on the theory of Markov chains and semi-Markov processes. The averaging scheme defines the deterministic trajectory of the mean workload, whereas the diffusion approximation scheme describes random deviations from this trajectory through an

Itô stochastic differential equation. This provides the foundation for predicting the behavior of an intelligent autoscaling system.

The dissertation has **also further developed and improved** the algorithm for parametric estimation and configuration optimization of a serverless system. Unlike existing approaches based on machine learning and neural networks, the proposed algorithm uses analytical estimates derived from queueing theory, enabling explicit formulas for optimal parameters and eliminating the need to train artificial intelligence models on large datasets.

An architectural framework for distributed data processing using serverless technologies on the AWS cloud platform **has been developed**. A distinctive feature of this architecture is its use of a message queue system, which enables loose coupling among the intelligent system's components, independent scaling when necessary, rapid recovery after failures, and more efficient use of computing resources. **An information technology** for distributed data processing based on serverless computing and proactive autoscaling of computing resources **has been developed and implemented**, forming the basis of the software created in this research.

The practical significance of the results lies in the implementation of an information technology for serverless computing optimization and software that provides workload prediction and proactive resource scaling on a cloud platform. Experimental validation on real-world datasets showed that the analytical model based on Markov chains and semi-Markov processes achieved a 25.8% increase in data processing speed, a 21.3% increase in throughput, and a reduction in cold starts to 3% compared with conventional reactive scaling. A comparative analysis also showed that the analytical model outperforms the machine-learning-based approach in the main performance indicators while providing quality-of-service guarantees without requiring large historical datasets.

The results of data mining and prediction obtained using the developed information technology can be used to optimize serverless computing in cloud environments. The developed software is an open framework that can be adapted

to different cloud platforms and used to build intelligent resource autoscaling systems based on the proposed models and algorithms.

The dissertation consists of an introduction, four chapters, a conclusion, a list of references, and four appendices.

The introduction substantiates the relevance of the research topic, formulates the aim, object, subject, objectives, and research methods, highlights the scientific novelty, and examines the connection of the work with existing scientific themes. It also specifies the author's personal contribution and provides information on the approval and publication of the main dissertation results. The structure and scope of the dissertation are described as well.

The first chapter presents the theoretical foundations and literature review in the field of serverless computing on cloud platforms. It examines the key concepts of the serverless computing paradigm, including the Function-as-a-Service and Backend-as-a-Service models, the architectural features of event-driven systems, mechanisms of automatic resource scaling, and integration with microservice architectures. The main advantages and limitations of serverless computing are analyzed, with particular emphasis on the cold-start problem, vendor lock-in, and the complexity of monitoring distributed systems. Metrics for evaluating the efficiency of serverless information systems are systematized, and open problems requiring further research and optimization are identified.

The second chapter is devoted to the analysis of architectural solutions for serverless computing and the development of a framework architecture for distributed data processing. It examines the role of event-driven architectures in building intelligent systems on cloud platforms and presents a comparative analysis of the main communication patterns. The choice of a message queue system as the core architectural component for asynchronous distributed data processing is justified by its ability to buffer requests and smooth peak workloads. A detailed comparative analysis of approaches to real-time communication is also provided. The limitations of reactive scaling, particularly cold starts and

unpredictable delays, are identified, which confirms the importance of developing proactive scaling algorithms based on workload prediction.

The third chapter develops a mathematical model of a serverless information system as a heterogeneous queueing system with mixed operating modes and proposes methods for estimating its key parameters. The properties of the task-stream mixture model, formed as a combination of independent non-homogeneous Poisson processes, are investigated, and its differences from the classical Markov-Modulated Poisson Process are established. A necessary and sufficient condition for the boundedness of the queueing system is established, and limiting results for the queue-length process under the averaging and diffusion approximation schemes are derived within the framework of semi-Markov random evolutions, based on the theory of Markov chains and semi-Markov processes. A method for estimating the parameters of the input-process mixture based on the EM algorithm and AWS platform metrics is developed. In addition, an algorithm for parametric estimation and optimization of the serverless system configuration is proposed. Unlike machine-learning- and neural-network-based approaches, it relies on analytical estimates, enabling explicit prediction of optimal parameters without requiring large training datasets.

The fourth chapter presents the practical verification of the theoretical results and the implementation of the proposed software for optimizing serverless computing on the AWS cloud platform. In the first experiment, predictive autoscaling based on DeepAR neural networks was studied; on real-world datasets, it reduced the number of cold starts by 27% and the number of unprocessed requests by 14%, while also revealing limitations of the approach, especially the need for large volumes of historical data for model training. In the second experiment, predictive autoscaling based on the analytical model with semi-Markov processes was investigated, resulting in a 25.8% increase in processing speed and a reduction in cold starts to 3%. In the third experiment, a Monte Carlo simulation of the intelligent system was carried out, confirming the advantages of the task-stream mixture model. The results of the data mining showed that the

analytical model outperforms the machine-learning-based approach in the main performance indicators, provides quality-of-service guarantees, and confirms the effectiveness of the developed information technology and software for serverless computing optimization.

The conclusions summarize the dissertation's main results.

The appendices contain scientific publications reflecting the main scientific results of the work, information on the approbation of the dissertation results, including acts and certificates of implementation, and a partial listing of the software code.

The proposed information technology for predictive autoscaling, which enables provisioning computing resources on a cloud platform in advance, is used by Finker Finance B.V. and Individual Entrepreneur S. I. Verbytska. The theoretical and practical results of the study are also used in the educational process of the Departments of Mathematical Problems of Control and Cybernetics and Software of Computer Systems at Yuriy Fedkovych Chernivtsi National University.

Keywords*:* information technology, cloud platform, optimization, model, algorithm, machine learning, neural networks, artificial intelligence, queuing system, intelligent system, data mining, Markov and semi-Markov chains, forecasting, dataset, software.

СПИСОК ПУБЛІКАЦІЙ ЗА ТЕМОЮ ДИСЕРТАЦІЇ

Наукові праці, в яких опубліковані основні наукові результати дисертації :

Наукові праці у виданнях, включених до переліку наукових фахових видань України та проіндексованих у наукометричній базі даних Scopus:

1. **Kyrychenko O.,** Ostapov S., Kyrychenko O. Design of a framework for serverless distributed data processing using queues. *Eastern-European Journal of Enterprise Technologies*. 2025. Vol.4, №9. P. 19–25. (**Scopus**) URL:

https://doi.org/10.15587/1729-4061.2025.335723 (*Особистий внесок: Кириченко О. О. – теоретична та практична розробка положень, розробка фреймворку, проведення дослідження, написання; Остапов С. Е. – постановка задачі, визначення загальної схеми дослідження, обговорення результатів; Кириченко О. Л. – аналіз літературних джерел, візуалізація, обговорення результатів*)

Наукові праці у періодичних наукових виданнях, проіндексованих у наукометричній базі даних Scopus:

2. **Kyrychenko O. O.**, Ostapov S. E., Kyrychenko O. L. Optimization of SQS Configurations for Efficient Batch Data Processing. *WSEAS Transactions on Systems.* 2025. Vol. 24. P. 36–43. (**Scopus**) URL: https://doi.org/10.37394/23202.2025.24.4 (*Особистий внесок: Кириченко О. О. – проведення дослідження, розробка хмарного прототипу, огляд літератури, написання, редагування рукопису; Остапов С. Е. – постановка задачі, концептуалізація; Кириченко О. Л. – валідація результатів дослідження, аналіз літературних джерел, написання, редагування*)

Наукові праці у виданнях, включених до переліку наукових фахових видань України:

3. Кириченко О., **Кириченко О**. Кешування даних у додатках з використанням безсерверної архітектури. *Information Technology: Computer Science, Software Engineering and Cyber Security*. 2024. №2. С. 42–49. URL: https://doi.org/10.32782/IT/2024-2-6 (*Особистий внесок: Кириченко О. О. – проведення дослідження, написання, редагування рукопису; Кириченко О. Л. – постановка задачі, керівництво дослідженням, рецензування та редагування*).

4. Кириченко О. Л., **Кириченко О. О.** Використання AWS APPSYNC для комунікації вебдодатків у реальному часі. *Вчені записки Таврійського національного університету імені В.І. Вернадського.* Серія : Технічні науки.

2024. 35(74), №4. С. 105–110. URL: https://doi.org/10.32782/2663-5941/2024.4/16 (*Особистий внесок: Кириченко О. О. – огляд літературних джерел, проведення дослідження, написання, редагування рукопису; Кириченко О. Л. – постановка задачі, визначення загальної схеми досліджень, обговорення результату*).

Наукові праці, які засвідчують апробацію матеріалів дисертації:

5. Антонюк Д. В., **Кириченко О. О.** Пакетна обробка даних на безсерверній архітектурі. *Проблеми інформатики та комп'ютерної техніки (ПІКТ–2023)* : праці XII праці Міжнар. наук.-практ. конф., м. Чернівці, 10–12 листопада 2023 р. Чернівці: Черн. нац. ун-т, 2023. С. 106–109. (*Особистий внесок: Кириченко О. О. – постановка задачі, методологія дослідження, проведення дослідження; Антонюк Д. В. – тестування, обговорення результатів, написання*).

6. Кириченко О. Л., **Кириченко О. О.** Покращення швидкодії безсерверних додатків за допомогою кешування. *Problems of Science and Technology: the Search for Innovative Solutions :* Proceedings of the XXIII International scientific and practical conference, Munich, Germany, 15–17 May, 2024. International Scientific Unity, 2024. P. 65–67. (*Особистий внесок: Кириченко О. О. – огляд літературних джерел, проведення досліджень, написання; Кириченко О. Л. – постановка задачі, редагування, рецензування)*.

7. **Kyrychenko O.**, Kyrychenko O. Real-time communication tools for web applications in a cloud environment. *The 13 th International Conference on Electronics, Communications and Computing's (IC ECCO)* : Materials of the Intern. Conf., Chisinau, Moldova, 17–18 October, 2024. P. 126–127. (*Особистий внесок: Кириченко О. О. – огляд літературних джерел, написання, рецензування та редагування; Кириченко О. Л. – постановка задачі, рецензування*).

8. **Кириченко О.О.,** Кириченко О. Л., Остапов С.Е. Аналіз продуктивності AWS SQS в умовах високих навантажень. *Проблеми*

інформатики та комп'ютерної техніки (ПІКТ–2024) : праці XIII Міжнар. наук.-практ. конф., м. Чернівці, 01–03 листопада 2024 р. Чернівці : Черн. нац. ун-т, 2024. С. 42–44. (*Особистий внесок: Кириченко О. О. – проведення досліджень, аналіз та опис результатів; Кириченко О. Л. – рецензування, обговорення результатів; Остапов С. Е. – постановка задачі, обговорення результатів*).

9. **Кириченко О**., Остапов С., Кириченко О. Безсерверний фреймворк із прогнозним масштабуванням на основі DeepAR для розподілених обчислень. *Trends and Prospects for the Development of Science and Education* : Proceedings of the 2nd International Scientific Conference (Oxford, United Kingdom, 10 July 2025). Lulu Press, Inc., 2025. P. 159–162. (*Особистий внесок: Кириченко О. О. – методологія дослідження, розробка програмного забезпечення, дослідження, написання; Остапов С. Е. – постановка задачі, загальне керівництво; Кириченко О. Л. – тестування, обговорення результатів*).

10. **Кириченко О.**, Малик І., Кириченко О. Оцінки довжини черги в неоднорідних системах масового обслуговування зі змінними режимами роботи. *Scientific Progress: Theories, Applications and Global Impact* : Proceedings of the 3rd International Scientific and Practical Conference (Braga, Portugal, March 2-4, 2026). European Open Science Space, 2026. P. 258–260. (*Особистий внесок: Кириченко О. О – аналіз та опис результату, написання; Малик І. В. – концептуалізація дослідження, аналіз результатів; Кириченко О. Л. – обговорення результатів, рецензування*).

Наукові праці, які додатково відображають наукові результати дисертації:

11. **Kyrychenko O.**, Ostapov S., Kyrychenko O. L. Predictive autoscaling in AWS Serverless by means of machine learning and SQS metrics. *CEUR Workshop Proceedings* : 6th International Workshop on Intelligent Information

Technologies & Systems of Information Security (IntelITSIS 2025, April 4, 2025). 2025. Vol.-3963. P. 77–87. (**Scopus)** URL: https://ceur-ws.org/Vol-3963/ (*Особистий внесок: Кириченко О. О. – проведення дослідження, тестування, написання; Остапов С. Е. – постановка задачі, концептуалізація; Кириченко О. Л. – обговорення результатів, рецензування*).

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

№	Скорочення	Розшифрування (англ.)	Переклад / Пояснення українською
1	AI	Artificial Intelligence	Штучний інтелект
2	API	Application Programming Interface	Інтерфейс програмування застосунків
3	AppSync	AWS AppSync	Керований сервіс GraphQL та комунікації в реальному часі AWS
4	AWS	Amazon Web Services	Хмарна платформа компанії Amazon
5	BaaS	Backend as a Service	Бекенд як сервіс
6	CloudWatch	Amazon CloudWatch	Сервіс моніторингу та спостереження AWS
7	CPU	Central Processing Unit	Центральний процесор
8	DeepAR	Deep AutoRegressive model	Модель глибокого авторегресійного прогнозування
9	DynamoDB	Amazon DynamoDB	Керована NoSQL база даних AWS
10	EM	Expectation-Maximization	Алгоритм максимізації очікування
11	EventBridge	Amazon EventBridge	Сервіс шини подій AWS
12	FaaS	Function as a Service	Функція як сервіс
13	FIFO	First In, First Out	Тип черги (перший прийшов – перший обслуговується)
14	HTTP	HyperText Transfer Protocol	Протокол передачі гіпертексту
15	IaaS	Infrastructure as a Service	Інфраструктура як сервіс
16	IAM	Identity and Access Management	Сервіс управління ідентифікацією та доступом AWS
17	IoT	Internet of Things	Інтернет речей
18	Lambda	AWS Lambda	Безсерверний обчислювальний сервіс AWS
19	MAE	Mean Absolute Error	Середня абсолютна похибка
20	ML	Machine Learning	Машинне навчання
21	MMPP	Markov-Modulated Poisson Process	Пуассонівський процес з марковською модуляцією
22	PaaS	Platform as a Service	Платформа як сервіс
23	PDF	Portable Document	Портативний формат документів

		Format	
24	QoS	Quality of Service	Якість обслуговування
25	RMSE	Root Mean Squared Error	Середньоквадратична похибка
26	RNN	Recurrent Neural Network	Рекурентна нейронна мережа
27	S3	Simple Storage Service	Сервіс об’єктного зберігання AWS
28	SaaS	Software as a Service	Програмне забезпечення як сервіс
29	SageMaker	Amazon SageMaker	Сервіс машинного навчання AWS
30	SLA	Service Level Agreement	Угода про рівень обслуговування
31	SLO	Service Level Objective	Цільовий показник рівня обслуговування
32	SQS	Simple Queue Service	Сервіс черг повідомлень AWS
33	WebSocket	WebSocket Protocol	Протокол двонаправленого зв’язку в реальному часі
34	CMO	Queueing system	Система масового обслуговування

ВСТУП

Актуальність та обґрунтування теми дисертаційного дослідження.

Розвиток хмарних технологій спричинив появу нових обчислювальних парадигм, таких яких безсерверні обчислення. Безсерверні платформи, зокрема AWS Lambda, Google Cloud Functions та Azure Functions, реалізують подійно-орієнтовану модель виконання, автоматичне масштабування ресурсів та модель оплати за фактичне використання, що значно спрощує процес розробки та розгортання хмарних додатків.

Незважаючи на значні переваги, безсерверні обчислення мають ряд обмежень, які знижують їх ефективність, особливо у розподілених системах з високим навантаженням. Серед таких обмежень найбільш критичними є проблема холодного старту, яка спричиняє додаткові затримки при першому виклику функції або після тривалого простою, механізми реактивного автоматичного масштабування обчислювальних ресурсів, що призводить до неефективного використання ресурсів при змінах у навантаженні, обмежений контроль для розробників над інфраструктурою хмарного провайдера, а також складність організації моніторингу та виявлення проблем у розподілених середовищах.

Сучасні наукові дослідження щодо оптимізації безсерверних обчислень в основному фокусуються на покращенні існуючих методів автоматичного масштабування, які здебільшого мають реактивний характер, тобто виділення додаткових обчислювальних ресурсів відбувається лише після виявлення перевищення наперед визначених порогових значень певних метрик. Підходи на основі прогнозування навантаження, хоча й забезпечують відповідну реакцію на зміни, але часто не враховують фактичну наявність ресурсів та особливості безсерверних архітектур, зокрема стохастичну природу вхідних потоків від різних незалежних джерел подій.

Досить часто безсерверні платформи доцільно розглядати як системи масового обслуговування. Це дозволяє моделювати процеси надходження завдань, їх обробки, перебування у черзі та автоматичного масштабування

ресурсів. Проте класичні моделі систем масового обслуговування (СМО) не повною мірою відображають складну стохастичну структуру безсерверних платформ, де вхідний потік формується як поєднання завдань з багатьох незалежних джерел з різними статистичними характеристиками, а процес обслуговування характеризується змінною інтенсивністю і залежить від конфігурації активних серверів.

Вагомий внесок у дослідження архітектур, методів оптимізації та моделювання безсерверних обчислень зробили іноземні та українські вчені. Серед фундаментальних робіт з аналізу безсерверних платформ та визначення відкритих проблем слід відзначити праці Baldini I. [1], Castro P. [2], Jonas E. [3] та Hellerstein J. [4]. Проблеми автоматичного масштабування та управління ресурсами в безсерверних середовищах досліджуються у працях Mampage A. [5], Buyya R. [6], Singhvi A. [7] та Gunasekaran J. [8]. Застосуванню безсерверних обчислень для ресурсомістких аналітичних задач та машинного навчання присвячені праці Nestorov A. M. [9], Carreira J. [10], García-López P. [11] та Wang H. [12, 13]. Питанням оптимізації холодного старту та прогнозування навантаження з використанням методів штучного інтелекту присвячені праці Nguyen T. [14], Golec M. [15] та Schuler L. [16]. Теоретичну основу для дослідження стохастичних процесів у системах масового обслуговування, що використовується в даній дисертації, закладено у роботах В. С. Королюка та І. В. Малика [17, 18]. Серед українських дослідників проблематику хмарних обчислень та безсерверних архітектур розглядали М. П. Шишкіна та І. А. Безвербний [19].

Таким чином, актуальність дисертаційного дослідження зумовлена необхідністю розробки нових математичних моделей та підходів до оптимізації безсерверних обчислень, які враховують неоднорідність вхідних потоків завдань, стохастичну природу часу виконання функцій та особливості автоматичного масштабування у хмарних середовищах. Вирішення цих проблем має практичне значення для підвищення продуктивності, зниження витрат та мінімізації затримок у безсерверних

системах.

Об’єктом дослідження є процеси розподіленої обробки даних у хмарних середовищах з використанням безсерверних технологій та систем керування чергами повідомлень.

Предметом дослідження є методи та моделі оптимізації безсерверних обчислень у хмарних середовищах на основі теорії систем масового обслуговування неоднорідної структури.

Метою дослідження є розробка підходів оптимізації безсерверних обчислень у хмарних середовищах, що забезпечують підвищення продуктивності, зниження витрат і мінімізацію холодних стартів шляхом побудови інформаційної технології на основі теорії масового обслуговування для прогнозування навантаження та проактивного масштабування обчислювальних ресурсів.

Для досягнення поставленої мети необхідно розв’язати такі завдання:

– провести аналіз сучасних підходів до реалізації, масштабування та оцінки ефективності безсерверних обчислень, визначити їх переваги та обмеження;

– побудувати математичну модель безсерверної обчислювальної системи як неоднорідної системи масового обслуговування зі змішаними режимами роботи, що враховує гетерогенність джерел подій та стохастичну природу процесів надходження та обробки завдань;

– розробити модель суміші потоків завдань у безсерверних обчисленнях, що формується як поєднання незалежних неоднорідних пуассонівських процесів надходження подій, та дослідити її відмінності від класичного Markov-Modulated Poisson Process (MMPP);

– дослідити граничні еволюції процесу довжини черги безсерверної системи у схемі усереднення та схемі дифузійної апроксимації з використанням апарату напівмарковських випадкових еволюцій, встановивши детерміновану траєкторію середнього навантаження та стохастичні відхилення від неї;

– встановити умови обмеженості черги та розробити алгоритм параметричної оцінки та оптимізації конфігурації безсерверної системи на основі аналітичних оцінок теорії масового обслуговування;

– розробити архітектуру фреймворку для розподіленої обробки даних з використанням безсерверної архітектури та модулем прогнозного автоматичного масштабування;

– реалізувати інформаційну технологію для розподіленої обробки даних з використанням безсерверних обчислень та проактивним автоматичним масштабуванням обчислювальних ресурсів;

– провести експериментальну апробацію розроблених моделей та інформаційної технології у хмарному середовищі AWS.

Методи дослідження.

Для розв’язання поставлених завдань у дисертації використано низку методів:

– методи теорії черг для побудови математичних моделей безсерверних обчислювальних систем, аналізу процесів надходження та обробки завдань, дослідження умов стабільності системи та оцінки основних параметрів продуктивності;

– методи теорії випадкових процесів та стохастичного аналізу для дослідження граничних еволюцій процесу довжини черги у схемах усереднення та дифузійної апроксимації;

– методи математичної статистики для оцінки параметрів моделей на основі метрик хмарних сервісів (AWS CloudWatch, Amazon SQS);

– методи оптимізації для визначення оптимальної конфігурації обчислювальних ресурсів неоднорідної безсерверної системи зі змішаними режимами роботи;

– методи порівняльного аналізу для дослідження архітектурних підходів до побудови безсерверних систем, способів комунікації між компонентами та визначення критеріїв вибору архітектурних рішень;

– хмарні технології для створення та тестування прототипів безсерверного додатку з використанням прогнозного та традиційного автоматичного масштабування обчислювальних ресурсів.

Наукова новизна одержаних результатів.

У дисертаційній роботі отримано такі нові наукові результати:

Вперше доведено граничні еволюції для процесу довжини черги у схемі усереднення та схемі дифузійної апроксимації з використанням апарату напівмарковських випадкових еволюцій. Схема усереднення визначає детерміновану траєкторію середнього навантаження Lambda-функції, а схема дифузійної апроксимації описує випадкові відхилення від цієї траєкторії через стохастичне диференціальне рівняння Ito. Встановлено необхідну та достатню умову обмеженості черги для неоднорідної СМО.

Набула подальшого розвитку математична модель безсерверної обчислювальної системи на прикладі AWS Lambda як неоднорідної системи масового обслуговування $H_k(t)/M/\infty$ зі змішаними режимами роботи, у якій вхідний процес визначається сумішшю незалежних неоднорідних пуассонівських процесів з різних джерел подій. На відміну від класичного Markov-Modulated Poisson Process (MMPP), запропонована модель враховує те, що кожне джерело (API Gateway, S3, SQS) генерує незалежний потік із власними статистичними характеристиками, що забезпечує більш адекватний опис дисперсії потоку завдань.

Удосконалено:

- підхід до оцінки кількості відхилених завдань (throttling rate) у безсерверних системах з обмеженою чергою на основі нормального наближення, що є узагальненням існуючих результатів для $M/M/\infty$ СМО з марковською модуляцією на випадок неоднорідного вхідного процесу зі змішаними режимами роботи та неперервним часом, що має практичне застосування для визначення відповідності вимогам Service Level Agreement (SLA);

- алгоритм параметричної оцінки та оптимізації конфігурації безсерверної системи, який на відміну від існуючих підходів на основі машинного навчання використовує аналітичні оцінки теорії масового обслуговування, дозволяючи отримати явні формули для оптимальних параметрів та уникнути необхідності навчання нейронних мереж.

Розроблено архітектуру фреймворку для розподіленої обробки даних з використанням безсерверних технологій. Особливістю даної архітектури є використання черги повідомлень, реалізація слабкої зв'язності між окремими елементами системи, незалежне їх масштабування в разі потреби, швидке відновлення після збоїв та оптимізація використання ресурсів.

Розроблено та реалізовано інформаційну технологію для розподіленої обробки даних з використанням безсерверних обчислень та проактивним автоматичним масштабуванням обчислювальних ресурсів.

Зв'язок роботи з науковими програмами, планами, темами. Дисертаційне дослідження виконано відповідно до планів науково-дослідницьких робіт:

– кафедри програмного забезпечення комп'ютерних систем Чернівецького національного університету імені Юрія Федьковича за держбюджетною тематикою: «Дослідження, моделювання та розробка програмного забезпечення складних динамічних систем» (Державний реєстраційний номер 0121U109232);

– кафедри математичних проблем управління і кібернетики Чернівецького національного університету імені Юрія Федьковича за держбюджетною тематикою: «Інформаційні технології в аспекті сучасних задач прийняття рішень» (Державний реєстраційний номер 0121U109159).

Публікації. За темою дисертації опубліковано 11 робіт, в яких подано результати теоретичних досліджень, алгоритмічних рішень та технологічної реалізації інформаційної системи для розподіленої обробки даних з

використанням безсерверних обчислень та проактивним автоматичним масштабуванням обчислювальних ресурсів. З них 4 статті у рецензованих виданнях (2 – в журналах, що індексується у наукометричній базі SCOPUS, 2 – в українських фахових виданнях), у матеріалах міжнародних наукових конференцій – 7 робіт (одна з них індексована в наукометричній базі Scopus).

Особистий внесок здобувача. Всі основні результати дисертаційного дослідження отримані автором самостійно. Зазначимо внесок автора у спільних працях: у праці [1] (Додаток А) Кириченку О.О. належить теоретична та практична розробка положень, розробка фреймворку, проведення дослідження, написання (Остапову С. Е. – постановка задачі, визначення загальної схеми дослідження, обговорення результатів; Кириченко О. Л. – аналіз літературних джерел, візуалізація, обговорення результатів); у праці [2] (Додаток А) Кириченку О.О. – проведення дослідження, розробка хмарного прототипу, огляд літератури, написання, редагування рукопису (Остапову С. Е. – постановка задачі, концептуалізація; Кириченко О. Л. – валідація результатів дослідження, аналіз літературних джерел, написання, редагування); у праці [3] (Додаток А) Кириченку О.О. належить проведення дослідження, написання, редагування рукопису (Кириченко О. Л. – постановка задачі, керівництво дослідженням, рецензування та редагування); у праці [4] (Додаток А) Кириченку О.О. – огляд літературних джерел, проведення дослідження, написання, редагування рукопису (Кириченко О. Л. – постановка задачі, визначення загальної схеми досліджень, обговорення результату); у праці [5] (Додаток А) Кириченку О.О. – постановка задачі, методологія дослідження, проведення дослідження (Антонюку Д. В. – тестування, обговорення результатів, написання); у праці [6] (Додаток А) Кириченку О.О. – огляд літературних джерел, проведення досліджень, написання (Кириченко О. Л. – постановка задачі, редагування, рецензування); у праці [7] (Додаток А) Кириченку О.О. – огляд літературних джерел, написання, рецензування та редагування (Кириченко О. Л. – постановка задачі, рецензування); у праці [8] (Додаток А)

Кириченку О.О. належить проведення досліджень, аналіз та опис результатів (Кириченко О. Л. – рецензування, обговорення результатів; Остапову С. Е. – постановка задачі, обговорення результатів); у праці [9] (Додаток А) Кириченку О.О. – методологія дослідження, розробка програмного забезпечення, дослідження, написання (Остапову С. Е. – постановка задачі, загальне керівництво; Кириченко О. Л. – тестування, обговорення результатів); у праці [10] (Додаток А) Кириченку О.О. – аналіз та опис результату, написання (Малику І. В. – концептуалізація дослідження, аналіз результатів; Кириченко О. Л. – обговорення результатів, рецензування); у праці [11] (Додаток А) Кириченку О.О. – проведення дослідження, тестування, написання (Остапову С. Е. – постановка задачі, концептуалізація; Кириченко О. Л. – обговорення результатів, рецензування).

Апробація результатів дисертації. Основні положення та наукові результати дисертаційної роботи доповідалися та обговорювалися на наукових семінарах кафедри програмного забезпечення комп'ютерних систем та кафедри математичних проблем управління і кібернетики, а також на Всеукраїнських та Міжнародних наукових і науково-практичних конференціях: «Проблеми інформатики та комп’ютерної техніки (ПІКТ)» (Чернівці, 2023-2024), «Problems of Science and Technology: the Search for Innovative Solutions» (Мюнхен, Німеччина, 2024); 13th International Conference on Electronics, Communications and Computing's (IC ECCO) (Кишинів, Молдова, 2024); «Trends and Prospects for the Development of Science and Education» (Оксфорд, Велика Британія, 2025); 6th International Workshop on Intelligent Information Technologies & Systems of Information Security (IntelITSIS 2025) (Хмельницький, 2025); «Scientific Progress: Theories, Applications and Global Impact» (Брага, Португалія, 2026).

Теоретичні та практичні результати дослідження впроваджені в роботі компанії Finker Finance B.V. та ФОП Вербицької С.І., а також, в освітньому процесі відділу комп’ютерних технологій Навчально-наукового інституту фізико-технічних та комп’ютерних наук Чернівецького національного

університету імені Юрія Федьковича, зокрема, в дисциплінах «Безсерверні обчислення у хмарних середовищах», «Проектування інформаційних систем».

Структура та обсяг роботи. Дисертаційна робота є завершеним дослідженням, загальним обсягом 245 сторінок (з них: 142 сторінки основного тексту, 26 сторінок – література, 47 сторінок – додатки). Дисертація складається зі вступу, чотирьох розділів із підрозділами, висновків, списку використаних джерел (194 позиції) та додатків.

РОЗДІЛ 1
ТЕОРЕТИЧНІ ОСНОВИ ТА ОГЛЯД ЛІТЕРАТУРИ

1.1 Безсерверні обчислення та їх ключові концепції

Поява та розвиток безсерверних обчислювальних платформ змінили загальний підхід до розгортання та масштабування додатків у хмарних середовищах. Вони розглядаються як продовження переходу від традиційних монолітних систем до віртуалізації, контейнеризації, мікросервісів та хмарно-орієнтованих архітектур [3]. У [1] безсерверні обчислення визначаються як модель виконання, у якій розробники повністю абстрагуються від управління інфраструктурою, зосереджуючись виключно на бізнес-логіці застосунку. Хмарні сервіси автоматично забезпечують масштабування, моніторинг та балансування навантаження, тоді як користувач сплачує лише за фактичний час виконання обчислень. Комплексні огляди еволюції безсерверних обчислень наведено в [2, 20, 21], де встановлено, що serverless є наступним етапом розвитку хмарних парадигм після IaaS, PaaS та контейнеризації. Перспективи подальшого розвитку безсерверних обчислень досліджено в [22].

Вирізняють два основні підходи в рамках парадигми безсерверних обчислень Function-as-a-Service (FaaS) та Backend-as-a-Service (BaaS). Function-as-a-Service (FaaS) – це модель виконання, у якій додаток реалізується у вигляді набору незалежних функцій, що активуються подіями або запитами. Кожен виклик функції виконується в ізольованому середовищі з ефемерним локальним станом, тобто стан системи між викликами повинен зберігатися у зовнішніх сервісах, таких як бази даних, сховища об’єктів тощо. Найбільш відомими прикладами реалізацій є AWS Lambda, Google Cloud Functions, Azure Functions.

Backend-as-a-Service (BaaS) – це модель надання керованого бекенду як послуги. Вона надає готові базові можливості, такі як автентифікація та керування доступом, збереження даних, файлове сховище, обробка даних в

реальному часі тощо. При такому підході розробники оперують абстракціями більш високого рівня, делегуючи хмарному провайдеру масштабування, резервування, оновлення та безпеку [23]. Огляд застосувань безсерверних обчислень у різних предметних областях, зокрема для Інтернету речей, наведено в [24], а аналіз FaaS з точки зору розробника – в [25].

Завдяки цим моделям безсерверні обчислення стали основою для побудови сучасних хмарних застосунків, орієнтованих на високу продуктивність, доступність, масштабованість та оптимальні витрати на експлуатацію [26].

1.1.1 Архітектурні особливості безсерверних обчислень

Характерною ознакою безсерверних платформ є їх подійно-орієнтована архітектура. Запуск та виконання бізнес-логіки ініціюється зовнішніми або внутрішніми подіями (HTTP-запити, повідомлення в чергах, таймери, тригери змін у сховищах). Такий підхід дозволяє створювати гнучкі й адаптивні застосунки, які масштабуються відповідно до змін у навантаженні системи [27].

Загалом архітектура безсерверних застосунків передбачає автоматичне виділення та вивільнення ресурсів. Безсерверні платформи оснащені контейнерними технологіями, які мають мінімальні затримки запуску, що дозволяє створювати тисячі екземплярів протягом кількох секунд. Якщо кількість запитів зростає, постачальник хмарних послуг автоматично створює нові екземпляри функцій, якщо ж навантаження зменшується то ресурси вивільняються. Це дозволяє уникнути проблеми надмірного або недостатнього використання обчислювальних ресурсів, що, як правило, трапляється в традиційних моделях обчислень [28]. Порівняльне оцінювання механізмів масштабування на платформах AWS Lambda, Azure Functions та Google Cloud Functions проведено в [29], а комплексний огляд продуктивності FaaS-платформ – у [30]. Дослідження [31] показало, що час запуску контейнерів суттєво залежить від мови виконання, обсягу

залежностей та конфігурації мережі. У багатьох дослідженнях автоматичне масштабування розглядається як один з основних принципів безсерверних обчислень. Зокрема, у роботі [32] зазначається, що без автоматичного масштабування безсерверні обчислення просто не можуть існувати та показує, що навіть у базовому вигляді без цієї функції неможливо забезпечити принципи оплати за використання (pay-per-use) та подійно-орієнтоване виконання.

Важливим аспектом є інтеграція безсерверних обчислень з мікросервісними архітектурами. Безсерверні функції можуть виступати як окремі компоненти обробки даних у розподілених системах, доповнюючи та розширюючи мікросервісний підхід. Це дозволяє ефективно розвантажувати основні мікросервіси від виконання допоміжних чи короткотермінових завдань [33, 34]. Завдяки цьому застосунки, побудовані з використанням мікросервісної архітектури отримують новий рівень модульності та гнучкості. Сервіси відповідають за довготривалу бізнес-логіку та управління станом, тоді як безсерверні функції дозволяють реалізувати легкі, швидко масштабовані й економічно ефективні компоненти. Така інтеграція спрощує розробку та оновлення систем, підвищує стійкість до навантажень, а також дозволяє більш точно розподіляти ресурси відповідно до реальних потреб користувачів та бізнесу [7]. Також, у деяких роботах, таких як [35], безсерверні функції розглядаються як окремі архітектурні шаблони, які інтегруються в мікросервісні системи. Вони можуть використовуватись як обгортки над мікросервісами, реалізуючи обробники специфічних подій, доповнюючи мікросервіси, а не замінюючи їх. Класифікацію та порівняльний аналіз існуючих FaaS-платформ наведено в [36], характеристику реальних безсерверних додатків – в [37], а стандартизовані набори тестів для оцінки FaaS – в [38–40].

Окремої уваги заслуговує концепція оркестрації функцій, яка є визначальною у реалізації взаємодії між незалежними безсерверними компонентами. Оркестрація дозволяє об’єднувати окремі функції у робочі

потоки з визначеною послідовністю виконання, встановлювати залежності, здійснювати контроль за станом і відслідковувати результати обчислень. На відміну від простого підходу, коли функція викликає функцію, оркестрація забезпечує централізоване керування взаємодією, обробку помилок, підтримку довготривалих процесів і транзакцій, що підвищує надійність і стійкість розподілених систем.

Так, у [1] підкреслюється, що модель “функція викликає функцію” має певні обмеження, які не дозволяють реалізувати складні бізнес-процеси, такі як довготривалі транзакції, керування помилками та стійкі залежності і призводять до проблем з масштабуванням обчислювальних ресурсів. Саме тому оркестрація є критичною для побудови складних безсерверних застосунків.

Дослідження [5, 41] описують оркестрацію як визначальну рису зрілих безсерверних застосунків, що дозволяє організувати взаємодію великої кількості дрібних функцій. Не менш важливою особливістю оркестрації є можливість автономного управління ресурсами, коли система самостійно масштабує потрібні функції у складі робочого потоку залежно від їхньої ролі.

Крім того, у [42] показується, що у складних наукових обчисленнях, оркестрація безсерверних функцій може виконувати роль повноцінних мікросервісів роблячи безсерверний застосунок більш гнучким.

Таким чином, оркестрація функцій стає не просто зручним, а й необхідним механізмом для організації розподіленої обробки даних, без якого неможлива реальна автоматизація управління безсерверними застосунками.

1.1.2 Основні переваги та обмеження безсерверних обчислень

Безсерверні обчислення мають набір переваг, які спрощують та пришвидшують розробку застосунків і дозволяють розглядати такий підхід як одну з основних архітектурних альтернатив при проєктуванні та створенні хмарних рішень.

Відсутність потреби в управлінні інфраструктурою є однією з головних відмінностей безсерверних обчислень від традиційних моделей розгортання. У такому разі розробники не мають потреби налаштовувати сервери, здійснювати оновлення операційних систем чи програмного забезпечення, забезпечувати балансування навантаження та організовувати моніторинг продуктивності. Усі ці завдання автоматично виконуються хмарним провайдером, що надає розробникам можливість зосередитися повністю на реалізації функціональності застосунку. В [3] автори називають таку відмінність як “abstraction of infrastructure” та зазначають, що зона відповідальності розробників обмежується кодом функцій залишаючи поза увагою управління інфраструктурою, це підвищує загальну продуктивність роботи. А у [43] підкреслюється, що під час переходу на безсерверні обчислення компанії фактично звільняються від обов’язку адмініструвати операційні системи, серверне середовище та оновлення інфраструктури, що суттєво знижує витрати на супровід та час виводу продуктів на ринок.

Ще однією перевагою безсерверних обчислень є автоматичне масштабування обчислювальних ресурсів у відповідь на зміну навантаження. На відміну від традиційних систем, де пік активності повинен прогнозуватися заздалегідь та відповідно резервуватися обчислювальні потужності, у безсерверній моделі виділення й звільнення ресурсів здійснюється повністю автоматизовано. Це означає, що кожен виклик функції отримує необхідний обсяг ресурсів для виконання, а після завершення обчислень вони одразу вивільняються. Так, в [7, 44] наголошується, що безсерверні обчислення відрізняються від класичних хмарних систем саме тим, що масштабування відбувається автоматично й прозоро для розробника. Таким чином, відсутня потреба у резервуванні обчислювальних ресурсів, система сама реагує на події та запити, кожен виклик функції виконується ізольовано та отримує ресурси на вимогу, які звільняються після виконання. З іншого боку, важливо уникати надлишкового використання обчислювальних ресурсів та витрат. Тобто,

виникає необхідність оптимального вибору конфігурацій, щоб досягти достатньо швидкого виконання, але без переплати за надмірні ресурси [45].

Але автоматичне масштабування має і низку проблем, таких як затримка при запуску нових екземплярів функцій, пошук оптимального співвідношення між вартістю і продуктивністю, оркестрація ресурсів при складних робочих потоках тощо. Ці проблеми потребують подальших досліджень.

Важливим принципом безсерверних обчислень є модель оплати за використання (pay-per-use), який змінює підхід до формування витрат на інфраструктуру. У цій моделі кошти сплачуються виключно за фактичний час виконання функцій та обсяг використаних ресурсів (пам’ять, обчислювальна потужність), а не за утримання серверів у постійно активному стані. Таким чином, відпадає потреба у попередньому резервуванні надлишкових ресурсів для пікових навантажень. Дослідження [41] визначає безсерверні обчислення не тільки як архітектурне рішення, а й як економічну парадигму, яка змінює спосіб використання хмарних ресурсів та повністю відрізняється від традиційних підходів із попереднім резервуванням віртуальних машин чи контейнерів. Завдяки автоматичному масштабуванню кожен виклик функції отримує ресурси саме у момент виконання, і оплата стягується лише за цей момент. Тобто витрати на інфраструктуру стають прогнозованими і прозорими, вартість прив’язана до кількості викликів і тривалості виконання функцій [46, 47]. У [48] запропоновано підхід до оптимізації вартості безсерверних обчислень через об'єднання функцій (function fusion), а в [49] – метод автоматичної конфігурації ресурсів функцій на основі статистичного навчання, що дозволяє зменшити витрати без погіршення продуктивності. Модель оплати за використання стає особливо актуальною у задачах де відсутнє постійне навантаження, таких як сервіси машинного навчання. У традиційних підходах для розгортання сервісів машинного навчання з використанням виділених серверів завжди є проблема простою обчислювальних ресурсів,

коли вони не використовуються через відсутність запитів або тренування моделі. На відміну від традиційних підходів безсерверні платформи дозволяють запускати обчислення при потребі й платити лише за фактично використані ресурси, що підвищує доступність сервісів машинного навчання [50].

Не менш важливою перевагою безсерверних обчислень є прискорення розробки хмарних застосунків. Це досягається завдяки інтеграції з сервісами Backend as a Service (BaaS) та використанню широкого набору готових хмарних рішень. Такий підхід дозволяє розробникам зосередитися на функціональності застосунку, мінімізуючи витрати часу на реалізацію інфраструктурних та допоміжних функцій. Його переваги підкреслюються в [51], де авторами запропонована платформа для аналітики даних у реальному часі на основі безсерверних обчислень і периферійних обчислень (edge computing), та в [52], де розглядається поєднання безсерверних обчислень і фог обчислень (fog computing) для роботи з непередбачуваним трафіком у реальному часі. Зазначені дослідження підтверджують той факт, що інтеграція з сервісами Backend as a Service (BaaS) дозволяє мінімізувати час розробки складних розподілених рішень, оскільки інфраструктурні питання (моніторинг, масштабування, управління навантаженням) виконуються автоматично. Автори вказують, що це особливо корисно у випадках з нестабільним та непередбачуваним трафіком, де традиційні підходи до планування ресурсів є економічно малоефективними.

Безсерверні обчислення мають також ряд недоліків та обмежень, які не сприяють застосуванню цієї парадигми у деяких випадках.

Проблема холодного старту є одним із суттєвих обмежень у безсерверних обчисленнях. Вона виникає під час початкового завантаження функції при першому виклику або після тривалого простою, коли обчислювальне середовище не зберігає активний стан функції. У такій ситуації хмарний провайдер змушений створити новий контейнер, підготувати середовище виконання, завантажити залежності та тільки після

цього запустити функцію. Це призводить до збільшення часу відгуку порівняно зі «звичайним» викликом, коли функція вже перебуває в активному стані. Дослідження [53] присвячене вивченню проблеми холодного старту, визначає її як одну з найбільш критичних обмежень безсерверних обчислень, особливо у системах реального часу, які потребують швидкої відповіді від сервісу. Роботи [7, 54] демонструють архітектурні та платформені оптимізації для зменшення холодного старту у високонавантажених сценаріях.

Ефект залежності від постачальника (Vendor lock-in) виникає тоді, коли застосунок або вся система тісно інтегровані з конкретним хмарним провайдером та його сервісами чи інструментами. У такому випадку перехід на іншу платформу стає технічно складним, дорогим і затратним по часу, що суттєво обмежує гнучкість бізнесу. Варто зазначити, що через відсутність стандартизації залежність від провайдера може стати головною відкритою проблемою безсерверних обчислень [1], оскільки економічні вигоди можуть нівелюватися високими витратами при перенесені системи на іншу хмарну платформу. Використання унікальних можливостей хмарного провайдера, оптимізація під конкретне середовище виконання, відсутність єдиних стандартів інтеграції та розгортання застосунків створюють фінансові та стратегічні ризики, а також ускладнюють довгострокове планування [56].

Відсутність впливу на налаштування продуктивності хмарного застосунку є одним із основних недоліків безсерверних обчислень, який випливає з самої сутності цієї моделі. Розробники отримують доступ лише до середовища для запуску функцій, тоді як управління інфраструктурою, включно з налаштуванням параметрів системи, залишається під повним контролем хмарного провайдера. Так, хмарний провайдер може встановлювати максимальні ліміти ресурсів доступних для використання, виконання функцій відбувається у спільному середовищі, що може мати негативний вплив на швидкодію через конкуренцію за обчислювальні ресурси. Також у [15, 56] стверджується, що відсутність доступу до

налаштувань ускладнює вирішення інших проблем безсерверних обчислень, таких як холодний старт та реактивне автоматичне масштабування. А у [45] зазначається, що оптимізація швидкодії безсерверних функцій повністю залежить від того, які можливості та інструменти надає хмарний провайдер. Тобто розробники вимушені пристосовуватись до рішень, які реалізовані у внутрішній інфраструктурі хмари. У [4] автори критично аналізують обмеження першого покоління безсерверних платформ, зокрема відсутність глобального стану та неможливість прямої адресації функцій через мережу. Прогнозування продуктивності та вартості безсерверних функцій на основі профілювання запропоновано в [57].

Безсерверні обчислення є складними з точки зору організації спостереження та відлагодження у розподілених середовищах. На відміну від монолітних або мікросервісних систем, де розробники мають ширший доступ до серверів, журналів та системних метрик, у безсерверній архітектурі значна частина інфраструктури повністю контролюється хмарним провайдером. Це обмежує можливості глибокого аналізу продуктивності та ускладнює пошук причин помилок та відмов у роботі. Безсерверна архітектура є своєрідною “чорною скринькою” для розробників, яка створює конфлікт, де з одного боку розробники не витрачають час на адміністрування інфраструктури, а з іншого – втрачають контроль над проблемами на системному рівні, що заважає аналізу та відлагодженю [3]. Відсутність доступу до інфраструктури змушує розробників використовувати непрямі методи моніторингу та створювати допоміжні механізми, що призводить до ускладнення процесу розробки та відлагодження хмарних застосунків [47, 58].

1.2 Відмінності безсерверних обчислень від традиційних обчислювальних моделей

Розвиток безсерверних обчислень супроводжується зміною підходів в управлінні інфраструктурою. Фізичні сервери, моделі віртуалізації, контейнеризації, сучасні безсерверні рішення визначають ступінь

відповідальності розробника за керування обчислювальними ресурсами, моніторинг і масштабування.

При розгортанні обчислювальних систем на фізичних серверах, вся відповідальність за налаштування, підтримку працездатності, оновлення та масштабування покладалася на системних адміністраторів. Така модель вимагала значних витрат часу та ресурсів, а також обмежувала гнучкість бізнесу.

Наступним етапом розвитку стала віртуалізація, яка дозволила ефективніше розподіляти та використовувати апаратні ресурси шляхом створення віртуальних машин (VM), що ізолювали різні середовища на одному фізичному сервері. Це зменшило витрати на обладнання, спростило управління обчислювальними ресурсами, однак вимагало певних зусиль для підтримки віртуальної інфраструктури.

Подальший розвиток у вигляді контейнеризації зробив можливим ще більш гнучке та спрощене розгортання застосунків. Контейнери надавали стандартизоване середовище виконання, спрощували масштабування та забезпечували сумісність між різними платформами. Проте навіть у контейнерних архітектурах зберігалася потреба у ручному або автоматизованому керуванні кластерами, налаштуванні мережевої взаємодії та моніторингу ресурсів (рис. 1.1).

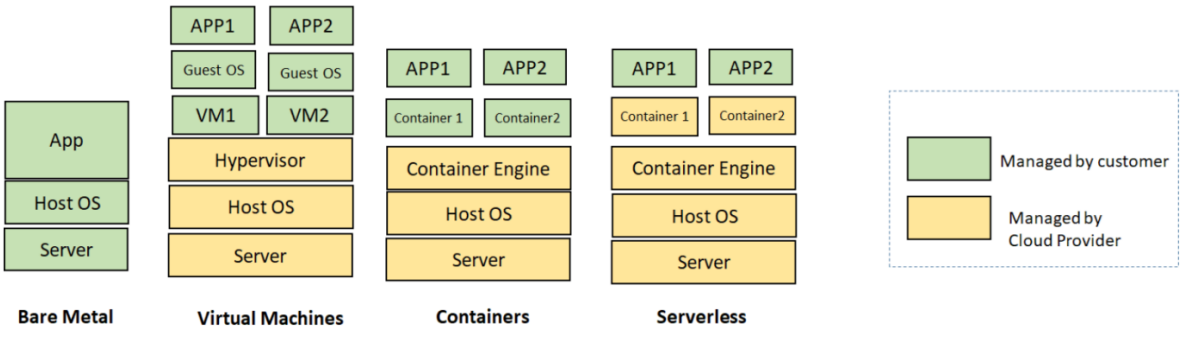

Рис.1.1 – Порівняння підходів в управлінні інфраструктурою

Поява безсерверних обчислень (serverless computing) стала логічним завершенням цієї еволюції. У даній парадигмі адміністрування

інфраструктури повністю делегується хмарному провайдеру. Натомість розробники можуть зосередитися виключно на розробці бізнес-логіки та функціональності додатків [23].

Загалом розвиток хмарних сервісів базувався на моделях, таких як: Інфраструктура як послуга (IaaS), Платформа як послуга (PaaS) та Програмне забезпечення як послуга (SaaS). Так, в [59] автори зазначають, що кожна модель має власну сферу застосування в залежності від необхідного рівня контролю над інфраструктурою.

Інфраструктура як послуга (IaaS) надає віртуалізовані апаратні ресурси та дозволяє розробникам отримати повний контроль над операційною системою та середовищем виконання. Разом з тим, на розробників покладається повна відповідальність за адміністрування, безпеку та масштабування [41]. Основними обмеженнями цієї моделі є високі витрати на утримання кваліфікованого персоналу, складність управління конфігураціями та підвищені ризики безпеки через необхідність самостійного налаштування захисту. Крім того, для IaaS характерна повільна адаптація до змін у навантаженні [60].

Платформа як послуга (PaaS) пропонує середовище для розробки та розгортання застосунків. Ця модель не передбачає для розробників управління серверами та налаштування параметрів середовища виконання на низькому рівні [5]. Суттєвим обмеженням PaaS є залежність від постачальника хмарних послуг (vendor lock-in), що значно ускладнює міграцію застосунків між різними платформами. У роботі [61] зазначається, що vendor lock-in є основною перепоною для впровадження єдиних стандартів у хмарних обчисленнях. Також втрачається гнучкість при виборі технологічного стеку та існує висока ймовірність погіршення продуктивності через відсутність індивідуальних конфігурацій в PaaS.

Програмне забезпечення як послуга (SaaS) відкриває доступ до повністю готової функціональності конкретного сервісу без необхідності мати справу з інфраструктурою. Головною ознакою даної моделі є

відсутність потреби у розробці програмного забезпечення. Однак, залишаються відкритими питання безпеки та конфіденційності даних. Зокрема у [41] програмне забезпечення як послуга описується як найвищий рівень абстракції у хмарному континуумі, де користувач отримує повністю готову функціональність без доступу до інфраструктури чи необхідності писати код. А у [2] зазначається, що ця модель знімає з користувача будь-які завдання з розробки й адміністрування. Разом з тим, SaaS надає мінімальний контроль над функціональністю. Користувачі стають залежними від політик провайдера хмарних послуг щодо оновлень, змін функціональності та цінової політики. Залишається відкритим питання використання чутливих даних через відсутність контролю над місцем зберігання та обробки даних.

Використання безсерверних обчислень призвело до появи ще однієї моделі – Функція як послуга (FaaS), яка повністю усунула потребу в управлінні інфраструктурою залишаючи при цьому можливість розробляти власний застосунок. FaaS передбачає виконання програмного коду у вигляді окремих функцій, що запускаються у відповідь на події, та усуває необхідність у налаштуванні середовища виконання на низькому рівні [1]. Також у [35] автори стверджують, що FaaS добре інтегрується у сучасні архітектури, оскільки функції можуть запускатися як окремі компоненти, замінюючи традиційні сервіси там, де потрібна гнучкість і масштабованість. Проте FaaS має низку обмежень. Найбільш критичною є проблема холодного старту, що робить цю модель недостатньо ефективною для застосунків реального часу. Дослідження [62, 63] показують, що холодний старт має негативні ефекти, включаючи затримки відповіді, зміни у продуктивності. Також важливим обмеженням є наявність максимального часу виконання функції, максимального розміру доступної пам’яті, що не дозволяє реалізовувати застосунки, які потребують довготривалих обчислень. Крім того, відсутність стану між викликами функцій ускладнює реалізацію складної бізнес-логіки. Варто зазначити, що складність відлагодження та моніторингу розподілених функцій також створює додаткові проблеми [53].

Безсерверні обчислення мають низку відмінностей від традиційних обчислювальних моделей.

Насамперед це управління інфраструктурою. На відміну від традиційних моделей, де розробники повністю або частково відповідають за управління середовищем виконання, в безсерверних обчисленнях інфраструктура прихована і є зоною відповідальності провайдера хмарних послуг. Так, відповідно до [41] саме прихованість інфраструктури та її повна залежність від провайдера є визначальною відмінністю FaaS від IaaS і PaaS.

Також важливою є модель обліку та витрат. Традиційні підходи передбачають оплату за резервування обчислювальних ресурсів незалежно від того чи будуть вони використовуватись чи ні. Безссерверні обчислення пропонують оплату за фактичне використання та сприяють зменшенню витрат при нерівномірному навантаженні [64].

Критичною відмінністю є масштабування обчислювальних ресурсів. На противагу ручним або напівавтоматичним налаштуванням процесу масштабування у системах, побудованих з використанням традиційних моделей, безсерверні обчислення пропонують повністю автоматичне масштабування. У [56] автори зазначають, що автоматичне масштабування є базовим механізмом безсерверних обчислень, без якого ця модель просто не може існувати, а в [44] стверджується, що виділення та вивільнення ресурсів без участі користувача кардинально відрізняє безсерверні обчислення від моделей віртуалізації, контейнеризації.

Не менш важливою відмінністю є контроль та продуктивність. Традиційні обчислення гарантують стабільну продуктивність із постійно працюючими серверами, що дозволяє уникати затримок холодного старту та мати повний контроль над оптимізацією. Продуктивність безсерверних обчислень в свою чергу повністю залежить від інфраструктури постачальника хмарних послуг та має недоліки у вигляді затримки холодного старту і обмеження часу виконання [45, 65]. Дослідження [58] демонструє, що холодні старти можуть спричиняти великі проблеми з продуктивністю

хмарних застосунків і пропонують різні підходи для вирішення цієї проблеми. А у роботі [1] автори визнають особливості контролю та продуктивності безсерверних систем одними із головних обмежень безсерверних обчислень. Вони звертають увагу на те, що у FaaS середовище не зберігається постійно у робочому стані, тому перший виклик функції супроводжується додатковими затримками та має часові обмеження на виконання, чого немає у традиційних моделях з постійно працюючими серверами.

Порівняння традиційних хмарних моделей IaaS, PaaS, SaaS та безсерверних обчислень FaaS проведено в таблиці 1.1.

Таблиця 1.1.

Порівняння традиційних хмарних моделей та безсерверних обчислень

Характеристика	IaaS (Інфраструктура як послуга)	PaaS (Платформа як послуга)	SaaS (Програмне забезпечення як послуга)	FaaS (Функції як послуга)
Керування інфраструктурою	Користувач управляє віртуальними машинами, мережею, ОС	Провайдер управляє платформою, користувач відповідає лише за код	Повністю управляється провайдером	Повністю прихована інфраструктура, користувач реалізовує функції
Масштабування	Ручне або автомасштабування ВМ	Вбудоване масштабування середовища	Масштабування залежить від провайдера	Автоматичне й прозоре масштабування функцій
Модель вартості	Оплата за виділені ресурси (CPU, RAM, диски)	Оплата за використання платформи	Оплата за підписку / ліцензію	Оплата за фактичне виконання

Контроль та продуктивність	Максимальний контроль над ресурсами	Баланс контролю та зручності	Мінімальний контроль	Мінімальний контроль з акцентом на продуктивність і швидкість розробки
Основні обмеження	Високі операційні витрати, складність управління, ризики безпеки	Vendor lock-in, обмежена гнучкість, потенційна неефективність при масштабуванні	Відсутність кастомізації, залежність від провайдера, проблеми з суверенітетом даних	Латентність холодного старту, архітектурні обмеження, складність відлагодження

Таким чином, FaaS можна розглядати як продовження IaaS, PaaS та SaaS, яке не тільки спрощує доступ до обчислювальних ресурсів, але й дозволяє інтегруватися з різноманітними хмарними сервісами. Водночас, традиційні моделі залишаються актуальними там, де потрібен контроль над конфігурацією, тоді як безсерверні обчислення є оптимальними для сценаріїв для подійно-орієнтованої обробки даних, швидкого створення прототипів та застосунків зі змінним навантаженням.

1.3 Оцінка ефективності безсерверних обчислень

Оцінка ефективності застосунків побудованих з використанням безсерверних архітектур вимагає нових підходів, оскільки традиційних метрик продуктивності, які використовуються для класичних хмарних моделей, не завжди достатньо для FaaS (Функція як послуга). На відміну від традиційних обчислювальних моделей, ефективність безсерверних систем визначається не лише класичними метриками продуктивності (час відгуку, пропускна здатність), а й специфічними характеристиками, що відображають

динаміку автоматичного масштабування, вартість використаних обчислювальних ресурсів та вплив різноманітних побічних ефектів.

1.3.1 Теоретичні моделі для аналізу ефективності

Одним з підходів для оцінювання ефективності безсерверних систем є використання теорії черг, яка часто застосовується для аналізу та прогнозування поведінки систем у розподілених архітектурах. У контексті безсерверних архітектур, ця теоретична база відіграє ключову роль у забезпеченні ефективної обробки даних у середовищах із високим рівнем навантаження. Для забезпечення масштабованості та оптимізації продуктивності в таких середовищах потрібні надійні моделі, оскільки швидкість надходження задач (λ) та швидкість їхньої обробки (μ) часто є непередбачуваними. Для спрощення висувається припущення, що події генеруються в системі відповідно до процесу Пуассона зі швидкістю λ і додаються в чергу по одному, а час обробки події розподілений експоненціально. Таке припущення дозволяє розглядати тільки поточний стан системи не звертаючи уваги на те в якому стані система була до того. Однак, на практиці час обробки подій не завжди розподілений експоненціально, подібним чином, час генерації подій також може не відповідати експоненціальному розподілу [66]. Це відповідно викликало потребу в модифікації класичних моделей, таких як *M/M/1* і *M/M/k*. Наприклад, у гібридних хмарних системах було запропоновано методи динамічної маршрутизації, які адаптуються під час виконання для максимізації ефективності обробки задач і мінімізації затримок у чергах [67] що підкреслює необхідну гнучкість для роботи з динамічними навантаженнями. Результати досліджень показали, що реалізація політик динамічної маршрутизації на основі теорії черг суттєво збільшує пропускну здатність і зменшує затримки. Такі досягнення вказують на потребу в адаптивних моделях, які можуть враховувати непередбачуваність

навантажень і забезпечувати еластичне масштабування, що є двома ключовими аспектами безсерверних платформ.

Важливими аспектами для систем, що проектувалися для роботи з високим навантаженням, є планування та пріоритизація. Дослідження [26, 68, 69] розглядали черги на основі пріоритетів, які прискорюють обробку завдань з високим пріоритетом та мінімізують затримки для критичних процесів. Ці методи стають дедалі більш актуальними в умовах робочих процесів, які поєднують пакетну обробку та обробку в реальному часі.

Основна концепція дослідження [70] полягає в оптимізації черг із механізмами пакетного обслуговування, де завдання обробляються у фіксованих за розміром групах, а не індивідуально. Ці системи суттєво відрізняються від класичних моделей черг, таких як *M/M/1*, оскільки вони передбачають пакетну обробку, яка впливає як на продуктивність системи, так і на її динаміку. Збільшення розміру пакетів може сприяти підвищенню пропускної здатності, але водночас може спричиняти довші часи очікування через затримки, пов’язані з формуванням пакетів.

Моделі черг є основою для аналізу та оптимізації розподілених, безсерверних програмних рішень у випадках наявності взаємодії між декількома компонентами. Одним з підходів для реалізації такої взаємодії є обмін повідомленнями, який може відбуватися як безпосередньо так і з використанням черг [71].

Сценарій із одним сервером, у якому один обчислювальний ресурс обробляє всі вхідні завдання, описується моделлю *M/M/1*.

Використання такого сервера визначається за формулою (1.1):

$$\rho = \frac{\lambda}{\mu}, \tag{1.1}$$

де $0 \leq \rho < 10$ забезпечує стабільність. Основні показники продуктивності:

- **середня довжина черги (L_q)**:

$$L_q = \frac{\rho^2}{1-\rho}; \tag{1.2}$$

- **середній час очікування (W_q)**:

$$W_q = \frac{\rho}{\mu(1-\rho)}; \tag{1.3}$$

- **середній час відповіді (W)**:

$$W = W_q + \frac{1}{\mu}. \tag{1.4}$$

Рівняння (1.2), (1.3), (1.4) демонструють, що L_q та W_q зростають експоненціально, що призводить до зниження продуктивності, коли ρ наближається до 1 (високе завантаження сервера).

Ця модель особливо актуальна для робочих процесів із низьким рівнем паралелізму в безсерверних середовищах, таких як обробка повідомлень з черги однією функцією невеликими пакетами. Модель дозволяє обчислювати завантаження сервера, довжину черги та час очікування, що є критично важливим для розуміння стабільності системи.

Модель *M/M/k* є узагальненням підходу з одним сервером для систем із кількома паралельними серверами, де обробка даних розподіляється між k серверами відповідно до формули (1.5):

$$\rho = \frac{\lambda}{k\mu}. \tag{1.5}$$

Основні показники включають:

- **Ймовірність, що всі сервери зайняті (P_{busy})**:

$$P_{busy} = \frac{\frac{\rho^k}{k!}}{\sum_{n=0}^{k}\frac{\rho^n}{n!}}. \tag{1.6}$$

Основою для цього значення є формула Erlang-B, яка обчислює ймовірність постановки завдань у чергу.

- **Середня довжина черги (L_q)**:

$$L_q = P_{busy}\frac{\rho}{1-\rho}. \tag{1.7}$$

- **Середній час очікування (W_q)**:

$$W_q = \frac{L_q}{\lambda}. \tag{1.8}$$

Модель *M/M/k* описує поведінку масивно-паралельних систем [72], де декілька функцій одночасно обробляють пакети повідомлень із черги. Вона особливо корисна для моделювання безсерверних робочих процесів, у яких функція автоматично масштабується для виконання паралельних завдань.

Ця модель дозволяє краще зрозуміти, як збільшення рівня паралелізму може підвищити пропускну здатність системи та зменшити час у черзі, що є критично важливим у періоди високого навантаження [73].

Для задоволення унікальних потреб безсерверних систем, окрім базових моделей, потрібні складніші концепції теорії черг. Наприклад, черги з пріоритетами часто використовуються для забезпечення швидшого виконання критичних завдань порівняно з менш важливими. Це особливо важливо для робочих процесів, які поєднують обробку в реальному часі та пакетну обробку. Так само, сама пакетна обробка безпосередньо впливає на завантаження сервера та довжину черги, масштабуючи швидкість надходження завдань залежно від розміру пакета (1.9):

$$\lambda_{effective} = b\,\lambda_{batch}\,. \tag{1.9}$$

Наприклад, швидкість надходження завдань прямо залежить від розміру пакета. Пакетна обробка збільшує пропускну здатність, однак окремі завдання в межах пакета можуть зазнавати затримок [74].

Загалом, безсерверні служби забезпечують високу масштабованість та економічну ефективність для сучасних додатків. Для динамічного налаштування параметрів, покращення продуктивності та оптимізації використання ресурсів було розроблено такі фреймворки, як FAASTloop, які вибирають відповідні параметри оптимізації за допомогою статистичних моделей [75]. Дослідження в хмарних середовищах продемонстрували здатність безсерверних архітектур ефективно обробляти робочі навантаження у великих масштабах [41]. Серед підходів до управління ресурсами в безсерверних середовищах слід відзначити: SLO-орієнтоване управління [8], платформу з низькою затримкою масштабування [7], гібридне планування для вартісно-ефективного виконання [76], автономне управління ресурсами

на основі метрик навантаження [77] та застосування навчання з підкріпленням для адаптивного автомасштабування [16]. Комплексний огляд методів автономного управління ресурсами в безсерверних обчисленнях, включно з методами на основі глибокого навчання з підкріпленням, наведено в [5].

1.3.2 Метрики ефективності безсерверних архітектур

Оскільки в безсерверних обчисленнях конфігурація та налаштування інфраструктури є зоною відповідальності постачальника хмарних послуг, то, відповідно, оцінка ефективності відбувається на основі власного набору критеріїв у порівнянні з традиційними хмарними моделями. Якщо у IaaS чи PaaS основними є показники завантаження процесора, використання пам'яті або коефіцієнт консолідації віртуальних машин, то у FaaS критичними є метрики, які безпосередньо впливають на якість сервісу та економічну ефективність використання. У науковій літературі та документації постачальників хмарних послуг виокремлюють такі групи метрик:

- *Продуктивність.*

Час відгуку (Response Time) – середній проміжок часу від моменту виклику функції до отримання результату обчислень. У багатьох наукових роботах дана метрика є базовим критерієм порівняння різних FaaS-платформ. Так, у [70] автори показали, що час відгуку безпосередньо впливає на якість обслуговування (QoS) і досвід кінцевих користувачів. Також у [1] вказується, що саме час відгуку, а не лише вартість чи модель виконання, визначає практичну придатність FaaS-платформи. Тому ця метрика активно використовується для порівняння різних безсерверних платформ [14]. А в [58] продемонстровано, що міграція між платформами часто ускладнена саме через різні показники часу відгуку і ця різниця критично важлива для користувачів.

- *Пропускна здатність*.

Пропускна здатність (Throughput) – кількість запитів, які система може обробити за одиницю часу. Вона визначає здатність безсерверної інфраструктури підтримувати високу інтенсивність викликів функцій без втрати стабільності. Зокрема, у роботі [1] зазначається, що FaaS повинна бути здатною обробляти велику кількість одночасних подій і тут пропускна здатність визначає наскільки добре інфраструктура може масштабуватися без втрати продуктивності. Дослідження [58] підкреслює важливість забезпечення стабільної пропускної здатності при пікових навантаженнях. Якщо функції не масштабуються достатньо швидко, система втрачає стабільність.

- *Затримки виконання.*

Однією з причин затримок у виконанні функції є ефект холодного старту (Cold Start), який виникає при першому виклику функції після періоду не активності [14]. Холодний старт призводить до збільшення часу виконання функцій, оскільки хмарний провайдер змушений створити новий контейнер, підготувати середовище виконання та підключити необхідні залежності [58]. Дослідження [15] ідентифікує проблему холодного старту як найбільш відому проблему продуктивності безсерверних обчислень, яка напряму пов’язана з додатковим часом завантаження середовища виконання та погіршує час виконання обчислень.

- *Надійність і відмовостійкість*.

Надійність і відмовостійкість безсерверних застосунків визначає якість сервісу та рівень довіри користувачів. На відміну від класичних моделей, де існує можливість контролю та налаштування механізму відновлення після збоїв, у FaaS-моделі відповідальність за відновлення сервісу здебільшого покладається на постачальника хмарних послуг, оскільки розробники не мають контролю над інфраструктурою [41].

Одним із ключових показників для оцінки надійності хмарних сервісів є ймовірність успішного виконання запиту (success rate). Цей показник демонструє, яка частина викликів завершується коректно. Так, у роботі [58]

наголошується, що для FaaS важливо вимірювати частку успішних викликів, оскільки навіть короткі відмови на рівні функцій можуть негативно впливати на тисячі паралельних запитів.

Ще однією важливою метрикою є середній час до відмови (Mean Time to Failure, MTTF), який відображає очікуваний період безперервного функціонування системи до появи критичної помилки. У [78] пояснюється, що ця метрика є основою для оцінки довготривалої стійкості сервісу при великих навантаженнях. А у [79] автори демонструють, що MTTF прямо впливає на продуктивність і відмовостійкість у масштабованих обчислювальних системах. Загалом, для безсерверних застосунків, що обробляють велику кількість даних, цей показник має особливе значення, адже навіть незначні відмови на рівні інфраструктури можуть масштабуватися й впливати на інші виклики функцій.

- *Вартість виконання.*

У безсерверних обчисленнях витрати розраховуються на основі оплати за використання [80]. На даний час більшістю постачальників хмарних послуг використовується однакова модель вартості виконання безсерверних функцій. В даній моделі вартість виконання залежить від часу, протягом якого функція знаходиться в активному стані, кількості обчислювальних ресурсів, які були виділені для функції, а також плати за кожен окремий виклик функції [23]. Однією з причин використання безсерверних обчислень є економія коштів та зниження операційних витрат, однак, на практиці досить складно оцінити їх розмір через зміну навантаження та затримку в отриманні статистичних даних від хмарного провайдера про кількість викликів та середню тривалість виклику протягом певного періоду часу [48].

У спрощеному варіанті вартість (1.10) визначається як

$$\mathrm{C} = \sum_{i=1}^{n}(t_i * p_i)\,, \qquad (1.10),$$

де t_i – час виконання окремої функції, а p_i – вартість виконання за одиницю часу. Однак в більшості випадків вартість також включає додаткові параметри, такі як кількість процесорного часу та виділеної оперативної

пам’яті. Так, у роботі [81] запропоновано використати підхід до розрахунку вартості виконання функцій із урахуванням кількох складових:

$$C_i = t_i * CPU * CPUCOST + t_i * MEMORY * MEMORYCOST + t_i * p_i ,$$

де

t_i – час виконання окремої функції,

p_i – вартість виконання за одиницю часу,

CPU – кількість виділеного процесорного часу,

CPUCOST – вартість одиниці процесорного часу,

MEMORY – кількість виділеної пам’яті,

MEMORYCOST – вартість пам’яті.

Такий підхід дозволяє значно покращити оцінку вартості, оскільки фінальні витрати залежать не лише від тривалості виконання окремих функцій, а й від характеру навантаження (рівномірне чи пікове), частоти викликів та оптимізації коду функції, зменшення його розміру чи використання більш ефективних алгоритмів. Крім того, на загальні витрати впливають і непрямі фактори, такі як ефект холодного старту, що збільшують середній час виконання, або неефективне використання пам’яті при розгортанні функцій. Тому дану метрику можна розглядати не тільки як економічний показник, але й як показник ефективного використання обчислювальних ресурсів.

- *Еластичність та масштабованість.*

Еластичність та можливість автоматичного масштабування обчислювальних ресурсів є характерною ознакою безсерверних обчислень, яка відображає здатність системи адаптуватися до змін у навантажені без втрати якості обслуговування. У традиційних обчислювальних моделях значні зміни у навантаженні потребували ручного втручання або попереднього резервування обчислювальних ресурсів, тоді як на безсерверних платформах ці процеси повністю автоматизовані.

Еластичність системи можна оцінити, використовуючи час реакції системи на зміну навантаження (Elasticity Reaction Time) та ступінь

відповідності виділених ресурсів фактичним потребам (Resource Matching Accuracy), а масштабованість визначається показниками пропускної здатності (Throughput) при зростанні кількості паралельних викликів та стабільністю часу відгуку (Response Time Stability) навіть при великому навантаженні [82].

Відповідно до [67] ефективність масштабування в безсерверних системах неможливо розглядати без врахування загальних витрат на обчислення. Автоматичне масштабування повинно забезпечувати баланс між продуктивністю та оптимальними витратами відповідно до моделі оплати лише за фактично використані ресурси. Надмірна кількість виділених обчислювальних ресурсів може призвести до зростання витрат, тоді як недостатня спричинити погіршення якості сервісу.

У роботі [44] також зазначається, що масштабування має забезпечувати баланс між продуктивністю та витратами, адже автоматичне масштабування може бути неефективним без контролю над затратами на виконання функцій.

Загалом, еластичність та масштабованість є важливими характеристиками, які дозволяють визначити, наскільки система здатна не лише підтримувати стабільність сервісу в умовах нерівномірного навантаження, але й робити це з мінімальними витратами обчислювальних ресурсів.

Висновки до розділу 1

У першому розділі розглянуто теоретичні основи та наведено основні підходи в рамках парадигми безсерверних обчислень.

Наведено архітектурні особливості безсерверних систем, серед яких визначальними є подійно-орієнтоване виконання функцій, автоматичне виділення та вивільнення ресурсів, інтеграція з мікросервісними архітектурами, а також концепція оркестрації функцій, що забезпечує побудову складних робочих процесів і підвищує надійність розподілених систем. Разом із тим встановлено, що відсутність контролю над

інфраструктурним рівнем породжує нові виклики, пов’язані з оптимізацією продуктивності, забезпеченням відмовостійкості та організацією спостереження за виконанням функцій.

Визначено ряд переваг безсерверних обчислень, зокрема: відсутність необхідності управління інфраструктурою, автоматизоване масштабування, скорочення фінансових витрат завдяки моделі оплати виключно за використані ресурси, а також прискорення процесів розробки завдяки інтеграції з широким спектром готових хмарних сервісів. Водночас виявлено й обмеження, серед яких найбільш значущими є ефект «холодного старту», залежність від постачальника (vendor lock-in), складність налагодження та моніторингу за відсутності доступу до інфраструктури.

Розглянуто основні підходи для оцінювання ефективності безсерверних систем та проведено аналіз ключових груп метрик, які дозволяють визначити, наскільки системи є стабільними в умовах нерівномірного навантаження. Також проаналізовано відкриті проблеми безсерверних обчислень, які потребують подальших досліджень, зокрема оптимізацію холодного старту, механізми автоматичного масштабування та забезпечення надійності.

Основні результати розділу опубліковані у працях [33, 34, 46, 65, 71, 82].

РОЗДІЛ 2
АРХІТЕКТУРНІ РІШЕННЯ ДЛЯ БЕЗСЕРВЕРНИХ ОБЧИСЛЕНЬ

Із зростанням популярності хмарних сервісів безсерверні обчислення стали однією з найкращих опцій при виборі архітектури в ході розробки хмарних застосунків. Можливість позбутися проблем, що пов’язані з управлінням інфраструктурою, автоматичним масштабуванням необхідних ресурсів та досягти необхідної ефективності використання коштів зробило безсерверні архітектури особливо привабливими при розробці додатків з розподіленими обчисленнями та високим навантаженням. Такий підхід дозволяє представити декомпозицію монолітного додатку низкою незалежних хмарних функцій. Розробники можуть концентруватися на реалізації бізнес-логіки залишаючи постачальникам хмарних послуг все інше. Саме тому безсерверні обчислення широко використовуються у розробці подієво-орієнтованих застосунків та додатків, в яких немає потреби зберігати стан безпосередньо в самому застосунку.

Досить часто в розподілених додатках виникає проблема налагодження оптимальної взаємодії окремих компонент, які можуть бути повністю незалежними і реалізованими за допомогою різних технологій та мов програмування. З цією метою розробники використовують черги повідомлень, які забезпечують асинхронний зв'язок між компонентами та буферизацію подій. Однак такий підхід не пропонує вбудованих механізмів для прогнозованого масштабування, що призводить до неефективного використання наявних ресурсів та збільшення загального часу виконання основної задачі. У зв’язку з цим наукові дослідження, спрямовані на розробку інструментів щодо динамічного масштабування обчислювальних ресурсів в залежності від робочого навантаження, залишаються надзвичайно актуальними. Результати таких досліджень важливі для практичного застосування, тому що дозволяють швидке розгортання розподілених систем

у хмарному середовищі, які здатні ефективно реагувати на зміни навантаження.

2.1 Архітектури на основі подій та їх роль в безсерверних обчисленнях

Архітектура, в основі якої лежить використання подій, є парадигмою для побудови сучасних розподілених систем, і вона відіграє основну роль у безсерверних обчисленнях. Така архітектура описує взаємодію окремих компонентів системи за допомогою обміну повідомленнями. З формальної точки зору подією є зміна стану системи, а повідомленням – нотифікація про подію. Системи, побудовані з використанням подійно-орієнтованої архітектури, як правило, складаються з компонентів, які генерують події та таких, що реагують на них.

Подібний підхід забезпечує слабку зв'язаність (loose coupling) між компонентами системи. Генераторам подій не потрібно знати про існування сервісів, що підписані на ці події та обробляють їх незалежно. Це значно спрощує розробку, тестування та масштабування окремих частин додатку.

Широке використання безсерверних обчислень (serverless computing) та сервісної моделі Function-as-a-Service (FaaS), зокрема AWS Lambda, Google Cloud Functions і Azure Functions, сприяло розвитку подійно-орієнтованої архітектури у хмарних середовищах. Функції у такій архітектурі запускаються лише за потреби – при настанні події, що значно покращує масштабованість системи і оптимізує вартість її використання [83]. Такий підхід дозволяє системі нарощувати потужність під час навантажень, гарантуючи швидкість відгуку та позитивне сприйняття продукту аудиторією. Набір джерел подій визначається хмарним провайдером. Кожне джерело може ініціювати виклик попередньо розгорнутих функцій відповідно до встановлених правил користувача. Типи підтримуваних джерел подій залежать від конкретної платформи й можуть включати HTTP-запити з інтерфейсу користувача, зміни у базах даних або файлових сховищах тощо.

Модель ціноутворення безсерверних рішень базується на принципі "оплата за фактичне використання", де тарифікація здійснюється виключно за фактичний час виконання коду та обсяг спожитих потужностей. Цей підхід значно рентабельніший за утримання традиційної інфраструктури, яка часто має надлишкові потужності, що простоюють. Економічна перевага рішення найбільш помітна в системах із варіативною або непередбачуваною інтенсивністю запитів. [28, 33].

Одним із прикладів реалізації подійно-орієнтованої взаємодії в безсерверних системах є використання сервісів для комунікації в реальному часі. У роботі [33] досліджено використання сервісу AWS AppSync, який за допомогою технології GraphQL Subscriptions дозволяє клієнтським додаткам підписуватися та отримувати оновлення даних (рис. 2.1).

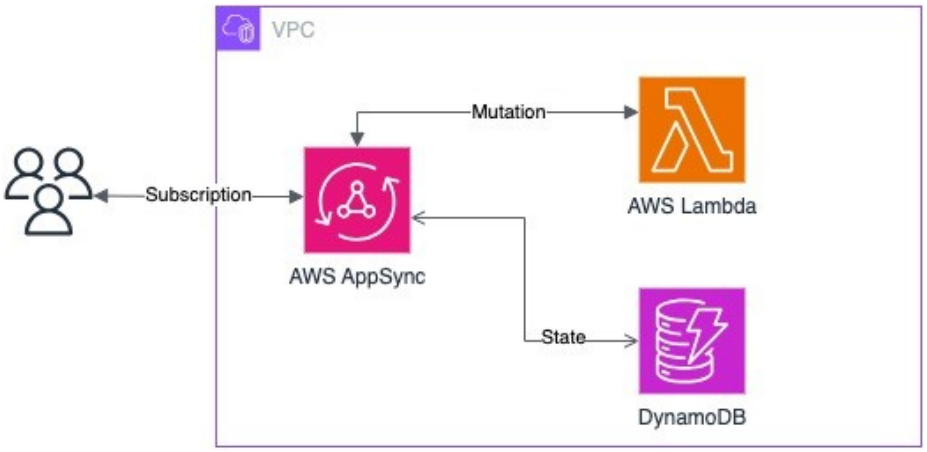

Рис. 2.1 – Архітектура вебсервісу для отримування інформації в реальному часі з застосуванням AppSync

Коли в джерелі даних змінюється стан, AppSync генерує подію і надсилає оновлені дані усім підписаним клієнтам одночасно. Завдяки цьому забезпечується миттєве інформування про фінальний статус операції – від успішного виконання до фіксації будь-яких збоїв. Схема взаємодії клієнта з AppSync наведена на рис. 2.2.

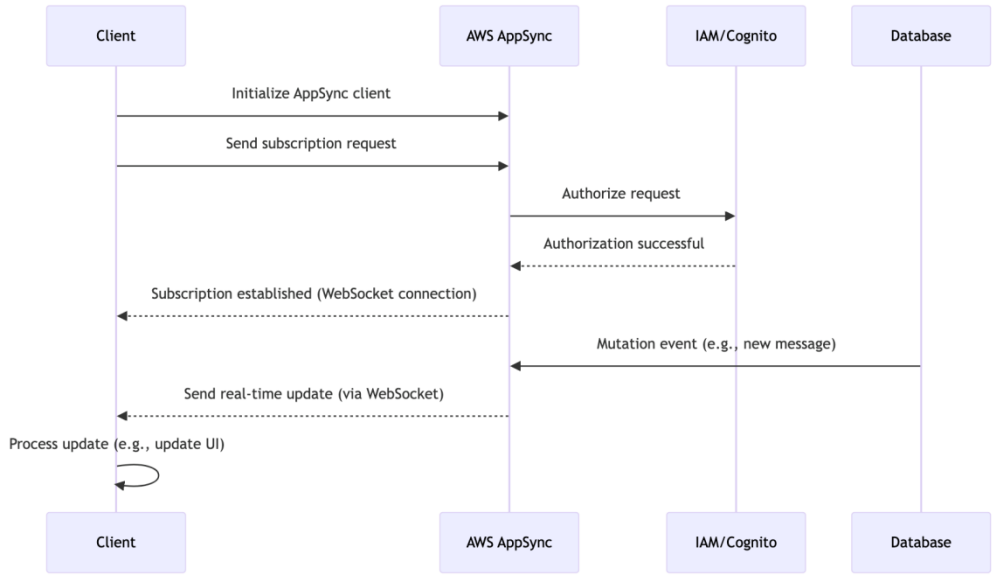

Рис. 2.2 – Схема взаємодії клієнта з AppSync

Варто зазначити, що AWS AppSync GraphQL захищено авторизацією IAM на сервері. Для надання доступу кінцевим користувачам використовується пул ідентифікаційних даних Amazon Cognito. AWS Lambda, яка відповідає за обробку бізнес даних, також використовує IAM роль для доступу до AppSync, що дозволяє уникнути несанкціонованого доступу до даних [33].

Подібний підхід є класичним прикладом подійно-орієнтованої архітектури, де AppSync відповідає за генерацію подій, а клієнтські додатки реагують на них.

Таким чином, подійно-орієнтовані архітектури є першим вибором при розробці безсерверних додатків. Вони дозволяють створювати гнучкі, масштабовані, високо адаптивні, економічно ефективні та стійкі до відмов розподілені системи.

2.2 Вибір безсерверних архітектур та їх порівняльний аналіз

Проєктування безсерверної архітектури – це процес пошуку балансу між продуктивністю, вартістю, надійністю та складністю розробки. Збільшення кількості шаблонів для розробки безсерверних додатків

викликало потребу в їх формалізованій класифікації та оцінюванні з погляду продуктивності, масштабованості, вартості та придатності для певних класів задач. Кожен шаблон має переваги та недоліки залежно від типу навантаження, вимог до збереження даних, взаємодії між компонентами. Огляд, проведений в [84], демонструє, що не існує універсального рішення і архітектура повинна бути адаптована до специфіки та вимог додатку.

2.2.1 Шаблони комунікації між компонентами

Вирізняють декілька способів, за допомогою яких функції та сервіси обмінюються даними. Наведемо деякі з них:

– *Запит-Відповідь (Request/Response)* [85].

У даному шаблоні один компонент надсилає запит іншому компоненту. Інший компонент обробляє запит та надсилає відповідь ініціатору комунікації. У безсерверних обчисленнях цей шаблон використовується для реалізації REST API за допомогою Amazon API Gateway. Основною його проблемою є ризик тайм-аутів при тривалих операціях обробки даних.

– *Публікація/Підписка (Publish/Subscribe)* [3, 86].

Це шаблон обміну повідомленнями, у якому відправники повідомлень не контактують безпосередньо з одержувачами. Повідомлення надсилаються посереднику (брокеру), який займається подальшою розсилкою відповідно до підписки. Брокер повідомлень відокремлює генератора подій від споживачів повідомлень за рахунок використання окремих каналів для кожної підписки. В AWS цей шаблон реалізується за допомогою Amazon SNS або Amazon EventBridge. Основними недоліками шаблону публікації-підписки є складність у пошуку проблем та моніторингу через асинхронність, труднощі з гарантією доставки та порядком повідомлень, ризик втрати повідомлень при недоступності підписників, потреба в ретельному налаштуванні безпеки та ускладнення тестування через відкладену обробку подій.

– *Черга повідомлень (Message Queueing)* [3, 87, 88].

Черга повідомлень є формою асинхронного зв'язку між сервісами, що використовується в безсерверних та мікросервісних архітектурах. Повідомлення зберігаються в черзі до моменту обробки та видалення. Кожне повідомлення обробляється лише один раз, одним споживачем. Черги повідомлень можна використовувати для буферизації або пакетної обробки даних, а також для згладжування пікових робочих навантажень. В AWS в якості черги повідомлень можна обрати Amazon Simple Queue Service (SQS). SQS дозволяє створювати повністю керовані черги повідомлень для мікросервісів, розподілених систем та безсерверних додатків. Основними недоліками Amazon SQS є обмеження на розмір повідомлень, відсутність гарантії порядку обробки у стандартних чергах, обмеження на термін зберігання повідомлень, а також складність у налаштуванні розширених функцій.

– *Шлюз API (API Gateway Pattern)* [3, 23, 89, 90].

Шлюз API є єдиною вхідною точкою для всіх клієнтів. Запити обробляються одним з двох способів. Деякі запити просто спрямовуються до відповідного сервісу. Інші запити обробляються за допомогою декількох сервісів. У безсерверних обчисленнях API Gateway виступає як рівень абстракції, що приховує деталі реалізації, такі як назви чи версії функцій, і може надавати додаткову функціональність таку як авторизацію, кешування даних, трансформацію запитів і відповідей не здійснюючі змін у самих функціях. До недоліків зазначеного шаблону можна віднести зростання архітектурної складності, ризик виникнення єдиної точки відмови, збільшення витрат на розробку та супровід.

2.2.2 Критерії вибору архітектури

Оптимальний вибір безсерверної архітектури повинен базуватися на системі критеріїв адаптованій під потреби конкретної задачі.

Проведемо порівняльний аналіз рішень для реалізації комунікації в реальному часі між сервісами на прикладі AWS AppSync та WebSockets через

API Gateway. Використання безсерверної архітектури для реалізації комунікацій у реальному часі надає значні переваги, що робить її привабливим варіантом для сучасних вебдодатків [33]. Так, використання таких сервісів, як AWS Lambda та API Gateway, дозволяє додаткам автоматично масштабуватися у відповідь на попит. Це означає, що під час пікових навантажень автоматично додаються необхідні ресурси, що забезпечує стабільну роботу та позитивний користувацький досвід. Така масштабованість є критично важливою для застосунків, що потребують оновлень у реальному часі, де попит користувачів може значно коливатися [33, 83].

Безсервені архітектури, у поєднанні з сервісами, такими як AWS AppSync або WebSockets через API Gateway, забезпечують канали комунікації з низькою затримкою. Зазначені сервіси гарантують, що оновлення надходять до користувачів у реальному часі, підвищуючи швидкість реагування та інтерактивність додатків. Затримка мінімізується завдяки використанню глобально розподіленої інфраструктури та граничних локацій [20, 33, 91-93].

Розглянемо простий вебсервіс, який є частиною системи для масової генерації pdf документів (рис. 2.3). Цей сервіс дозволяє в реальному часі отримувати інформацію стосовно статусу чергової генерації [33].

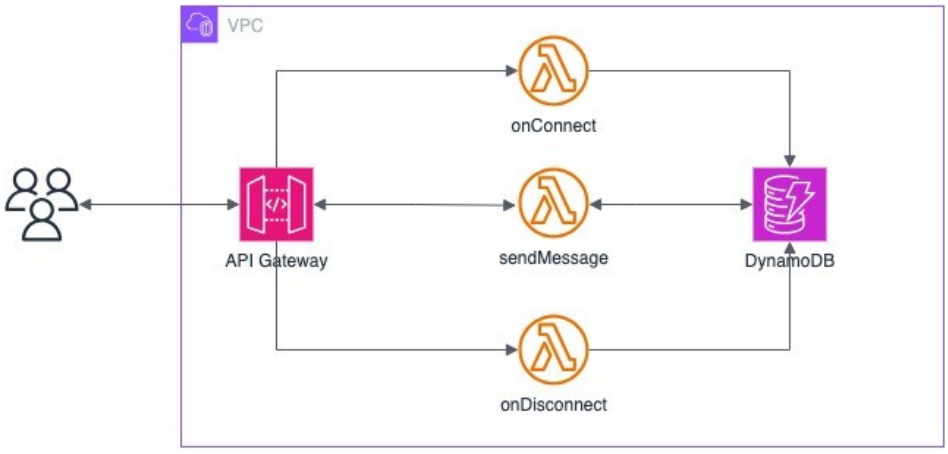

Рис. 2.3 – Архітектура вебсервісу для обміну інформацією в режимі реального часу

Архітектура вебсервісу побудована на взаємодії AWS Lambda з Amazon API Gateway та Amazon DynamoDB. У цій екосистемі API Gateway відповідає за безпечну публікацію та комплексне управління API-інтерфейсами [94]. Зі свого боку, DynamoDB забезпечує високу продуктивність системи як повністю керована безсерверна NoSQL-база даних, що працює за принципом «ключ-значення» і здатна витримувати будь-які навантаження [95].

На основі API Gateway можна створити WebSocket API зі збереженням стану з’єднання в DynamoDB. Такий підхід дозволить WebSocket API викликати інші сервіси на основі вмісту повідомлень, які надходять від клієнтів. На відміну від REST API, який отримує запити та відповідає на них, WebSocket API забезпечує двосторонній зв'язок між клієнтськими додатками та серверною частиною. Це означає, що сервер може надсилати повідомлення безпосередньо підключеним клієнтам [96, 97].

Іншим підходом реалізації комунікації в реальному часі з застосуванням безсерверних технологій є використання AWS AppSync [33, 98]. AWS AppSync використовує переваги підписки GraphQL для виконання операцій у реальному часі, надаючи дані клієнтам, які вирішують прослуховувати певні події з серверної частини. Це означає, що можна без особливих зусиль створити будь-яке підтримуване джерело даних у режимі реального часу в AWS AppSync, а керування з’єднанням автоматично оброблятиметься клієнтом AWS AppSync, використовуючи WebSocket як мережевий протокол між клієнтом і сервісом (рис. 2.4) [33, 98, 99].

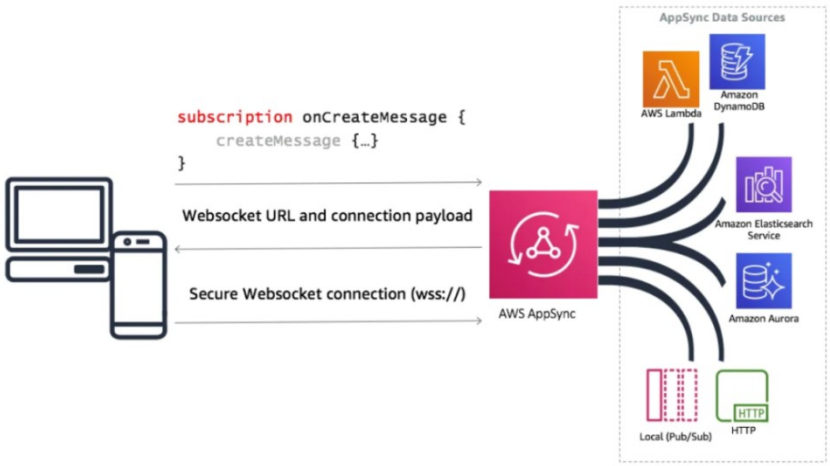

Рис. 2.4 – Підписки на GraphQL у реальному часі в AppSync [99]

Порівняння здійснюватиметься на основі наступних критеріїв:

- *Простота розгортання* – Кількість конфігурацій, легкість масштабування.
- *Підтримка реального часу* – Затримка доставки повідомлень, стабільність з'єднання.
- *Масштабованість* – Підтримка тисяч одночасних з'єднань.
- *Підтримка авторизації та безпеки* – Наявність вбудованої підтримки IAM, Cognito.
- *Обробка стану* – Збереження стану з'єднання.
- *Вартість* – Вартість сервісу за певну кількість або запитів.

Результати проведеного порівняльного аналізу наведені в таблиці 2.1.

Таблиця 2.1.

Порівняльний аналіз за обраними критеріями

Критерій	AppSync	API Gateway (WebSocket)
Простота розгортання	Автоматичне керування WebSocket, конфігурація через GraphQL schema	Потрібне ручне налаштування

Підтримка реального часу	<150 мс при кількості заптитів менше 1000	<120 мс при кількості заптитів менше 1000
Масштабованість	До 100 тис. підписників через GraphQL endpoint	До 500 тис. підключень, але потребує додаткового моніторингу
Підтримка авторизації та безпеки	Вбудована підтримка Cognito, IAM	Потребує окремої реалізації
Обробка стану	Контекст прив'язується до кожного клієнта	Потрібно ручне керування з’єднаннями
Вартість	$4 на 1 млн. запитів + платні підключення	$1–2 на 1 млн. повідомлень

Таким чином, основними рекомендаціями для вибору AWS AppSync при розробці хмарного додатку для реалізації комунікації в реальному часі є [33]:

- можливість використання GraphQL;
- необхідність швидкої інтеграції з іншими сервісами AWS (DynamoDB, Lambda, Cognito);
- наявність автоматичного масштабування;
- наявність вбудованої підтримки AWS IAM, Cognito та інших механізмів автентифікації і авторизації.

Остаточний вибір між AWS AppSync та API Gateway WebSocket залежить від особливостей вимог та архітектури додатку для отримування інформації в реальному часі, таких як складність запитів, вимоги до масштабованості, вартість та рівень необхідної інтеграції з іншими сервісами [33]. Використання безсерверної архітектури для реалізації комунікації у реальному часі дозволяє створювати надійні, масштабовані та економічно

ефективні системи, які відповідають вимогам сучасних вебдодатків, забезпечуючи високу продуктивність та задоволення користувачів. Це робить безсерверні компоненти ключовими елементами у створенні ефективних рішень для комунікацій у реальному часі [33].

Загалом, вибір архітектури це складний процес, який вимагає значної кількості компромісів. Складні системи вимагають гібридного підходу коли в одному рішенні поєднуються декілька різних шаблонів, що дозволяє використати переваги кожного з них.

2.3. Управління ресурсами та динамічним масштабуванням

Безсерверні обчислення стали однією з найкращих опцій при виборі архітектури в ході розробки хмарних додатків. Можливість позбутися проблем, пов’язаних з управлінням інфраструктурою, автоматичним масштабуванням необхідних ресурсів та досягти необхідної ефективності використання коштів зробило безсерверні архітектури особливо привабливими при розробці додатків з розподіленими обчисленнями та високим навантаженням. Такий підхід дозволяє зробити декомпозицію монолітного додатку на низку незалежних хмарних функцій. Розробники можуть концентруватися на реалізації бізнес-логіки залишаючи постачальникам хмарних послуг все інше. Саме тому безсерверні обчислення широко використовуються у розробці подієво-орієнтованих додатків та додатків де немає необхідності зберігати стан безпосередньо у самому додатку.

Незважаючи на суттєві переваги, безсерверні обчислення мають ряд обмежень, що значно ускладнює процес обробки даних у розподілених середовищах. Насамперед це лімітований час виконання хмарної функції, що вимагає розбиття основної задачі на незалежні підзадачі відповідного об’єму та координації обробки даних в підзадачах. Ще одним обмеженням є залежність будь-якого безсерверного додатку від механізмів реактивного масштабування, коли додаткове виділення ресурсів відбувається як відповідь

на зміну у навантаженні [55]. Це, в свою чергу, призводить до збільшення кількості холодних стартів та проблем із затримкою відповіді. Окрім того, відсутність стану в безсерверних додатках ускладнює відслідковування прогресу виконання окремих завдань в режимі реального часу.

Досить часто в розподілених додатках виникає проблема налагодження оптимальної взаємодії окремих компонент, які можуть бути повністю незалежними і реалізованими за допомогою різних технологій та мов програмування. З цією метою розробники використовують черги повідомлень, які забезпечують асинхронний зв'язок між компонентами та буферизацію подій. Однак такий підхід не пропонує вбудованих механізмів для прогнозованого масштабування, що призводить до неефективного використання наявних ресурсів та збільшення загального часу виконання основної задачі.

Для підвищення ефективності управління ресурсами застосовують два принципово різні підходи – реактивне та проактивне масштабування.

При реактивному масштабуванні система реагує на зміну навантаження та виділяє додаткові ресурси для обробки даних. Незважаючи на економічну ефективність зазначеного підходу, він є першопричиною холодних стартів при зростанні трафіку. Для подолання подібних обмежень в розподілених системах є пакетна обробка. В роботі [53] запропоновано методи, які дозволяють об’єднувати декілька задач в пакети та обробляти їх як єдине ціле. Такий підхід сприяє зменшенню кількості холодних запусків функції та підвищує загальну ефективність системи особливо за умов високого навантаження.

Важливим підходом до вирішення проблем реактивному масштабування безсерверних додатків є використання можливостей Amazon Web Services (AWS) Simple Queue Service (SQS) [56, 100]. У такому випадку тригером для початку масштабування є зміна ключових характеристик черги, таких як кількість повідомлень, що очікують на обробку, або кількість повідомлень, що вже оброблені. Варто зазначити, що правильно налаштоване

реактивне масштабування призводить до зменшення кількості холодних стартів в додатку. Традиційні методи для реактивного масштабування ресурсів базуються на порогових значеннях та в цілому демонструють непогані результати, однак відчувають певні труднощі у випадках раптового збільшення навантаження. Звичайно хмарний провайдер може тримати підготовлені, наперед прогріті контейнери для згладжування пікового навантаження, але це не завжди економічно доцільно [44].

Метою проактивного динамічного масштабування обчислювальних ресурсів є підготовка ресурсів до того, як вони знадобляться. Замість того, щоб реагувати на фактичні зміни у навантаженні, система намагається передбачити майбутню потребу і заздалегідь виділити необхідну кількість екземплярів функцій. Це основний метод боротьби з холодними стартами.

Досить часто пропонується застосування прогнозного автоматичного масштабування з використанням моделей машинного навчання. Завдяки такому підходу можна завчасно планувати використання обчислювальних ресурсів [101]. Однак досить часто методи для прогнозованого автоматичного масштабування не враховують фактичну наявність ресурсів особливо у додатках, які використовують черги, тобто виникає потреба у моніторингу параметрів черг для прийняття рішення стосовно масштабування [102].

Незважаючи на значну кількість досліджень, присвячених оптимальному використанню ресурсів при розподілених обчисленнях у хмарних середовищах, підходи, специфічні саме для безсерверних архітектур, залишаються недостатньо вивченими. Більшість обмежуються реактивними методами або вирішують часткові аспекти. Особливості розподілу відповідальності між хмарним провайдером та розробниками не дозволяють повноцінно контролювати поведінку додатку та ускладнюють пошук оптимальних конфігурацій. Відсутність повної інформації стосовно управління ресурсами хмарним провайдером ускладнює проведення досліджень та проєктування безсерверних рішень.

Враховуючи реактивність традиційних підходів для автоматичного масштабування, що призводить до затримок під час раптової зміни навантаження в розподілених системах з використанням черг, розроблено фреймворк для прогнозування навантаження та превентивного масштабування ресурсів [103].

При розробці фреймворку:

– розроблено його архітектуру;

– реалізовано модуль прогнозного автомасштабування для оцінки ймовірності переходу системи між різними станами навантаження та відповідної зміни конфігурації черг і обробників даних.

Гіпотеза дослідження передбачає, що використання безсерверних архітектур у поєднанні з чергами повідомлень (зокрема AWS SQS) дозволяє ефективно автоматично масштабувати розподілену обробку даних, зменшити час відповіді системи, кількість холодних запусків та підвищити загальну продуктивність. Такий підхід реалізує слабку зв’язність окремих елементів системи та забезпечує масштабування як системи в цілому так і окремих її частин в разі потреби. При проведенні моделювання зроблено наступні припущення:

– всі повідомлення, що надходять у чергу, є валідними, мають однаковий розмір;

– масштабування обробників даних здійснюється в межах квоти, наданої хмарним провайдером, і не залежить від внутрішніх обмежень хмарної платформи;

– всі компоненти системи розгорнуті в межах одного регіону AWS для мінімізації впливу затримок у мережевих з’єднаннях.

Методом дослідження є експериментальний підхід, який передбачає створення та тестування прототипів безсерверного додатку для розподіленої генерації PDF документів з використанням прогнозного та традиційного автомасштабування та подальше порівняння отриманих результатів.

Опишемо функціональні та нефункціональні вимоги до фреймворку. Слід зазначити, що наявність чітко визначених функціональних та нефункціональних вимог є критичною для успішної розробки безсерверних додатків. Безсерверні додатки функціонують у хмарному середовищі з обмеженим доступом до внутрішньої інфраструктури і розробники не керують інфраструктурою чи розгортанням. Тому без чітко визначених вимог складно гарантувати очікувану поведінку системи та неможливо прогнозувати ресурсне навантаження [104, 105].

Функціональні вимоги:

- Система повинна забезпечувати надійну, масштабовану та асинхронну обробку даних.
- Необхідно реалізувати надання кінцевим користувачам інформацію про стан обробки їхніх завдань у режимі реального часу.
- Передбачено динамічну оптимізацію власних ресурсів на основі прогнозованого навантаження для досягнення максимальної ефективності.
- Використовується черга повідомлень для буферизації та асинхронної передачі завдань на рівень обробки.
- Логіка обробки даних покладається на функції AWS Lambda (Processing Lambda).
- Функції обробки мають автоматично активуватися при появі нових повідомлень у черзі SQS.
- Програмне рішення повинно підтримувати одночасний запуск однієї або декількох функцій обробки залежно від кількості повідомлень та конфігурації.
- Система зберігає проміжний та кінцевий результати кожного завдання обробки.
- Як сховище для стану обрано базу даних AWS DynamoDB.

- Записи про стан повинні містити достатньо інформації для відстеження прогресу виконання та ідентифікації можливих помилок.
- Зміна стану завдання в DynamoDB автоматично ініціює сповіщення.
- Підключені клієнти отримують оновлення статусу обробки своїх завдань у реальному часі без необхідності робити повторні запити.
- Алгоритм передбачає запуск спеціальної функції за розкладом для отримання прогнозу навантаження.
- На основі отриманого прогнозу система динамічно коригує конфігураційні параметри сервісів з метою оптимізації використання ресурсів.

Нефункціональні вимоги:

- Цільову пропускну здатність архітектури розраховано на обробку не менше однієї тисячі повідомлень на хвилину відповідно до очікуваного навантаження.
- Граничний час очікування від моменту оновлення стану завдання в DynamoDB до моменту отримання сповіщення клієнтом через AppSync обмежений 300 мілісекундами за нормальних умов.
- Мінімізація затримок, спричинених холодним стартом, забезпечується прогнозним автомасштабуванням для 95% запитів під час пікових навантажень.
- Цільовий рівень доступності встановлено на позначці не нижче 99.9%, що досягається завдяки використанню високонадійних керованих сервісів AWS.
- Гарантується цілісність даних: у разі триразової невдалої спроби обробки завдання повідомлення автоматично переміщується до черги "мертвих" повідомлень (DLQ) для подальшого аналізу.

– Проєктування компонентів як безстанових дозволяє платформі AWS автоматично перезапускати їх у разі збоїв без втрати прогресу обробки.
– Кількість функцій-обробників автоматично адаптується (масштабується вгору та вниз) залежно від довжини черги SQS.
– Проактивне регулювання кількості «теплих» екземплярів Lambda базується на моделі SageMaker, що дозволяє випереджати сплески навантаження без затримок.
– Оптимізація витрат досягається шляхом динамічного звільнення невикористовуваних ресурсів у періоди низької активності.
– Архітектура дотримується принципу «оплата за фактичне використання», що виключає витрати на ресурси, які простоюють.
– Слабкий зв'язок між рівнями фреймворку уможливлює їхню незалежну модифікацію та розгортання.

2.4. Фреймворк для розподіленої обробки даних з використанням безсерверної архітектури

Архітектура фреймворку використовує наступні сервіси AWS:

– AWS S3 для зберігання вхідних даних та результатів обчислень [106];

– Amazon SQS в якості черги завдань [107];

– AWS Lambda для обчислень на основі подій [108];

– DynamoDB для збереження стану обробки даних [95];

– AWS AppSync для моніторингу стану обробки даних в реальному часі [98];

– Amazon SageMaker для розгортання моделі прогнозування навантаження [109];

– AWS EventBridge в якості планувальника для запуску AWS Lambda за розкладом [110].

Загальна архітектура фреймворку представлена на рис. 2.5 та побудована за багаторівневим принципом, де кожен рівень має чітко визначену зону відповідальності. Такий підхід дозволяє забезпечити необхідну ізоляцію окремих компонентів та відповідно покращити масштабованість системи, спростити її розробку та супровід, а також полегшити тестування. Основними рівнями фреймворку є:

– рівень завантаження даних;

– рівень обробки даних;

– рівень управління станом;

– рівень моніторингу в реальному часі;

– рівень прогнозного автомасштабування.

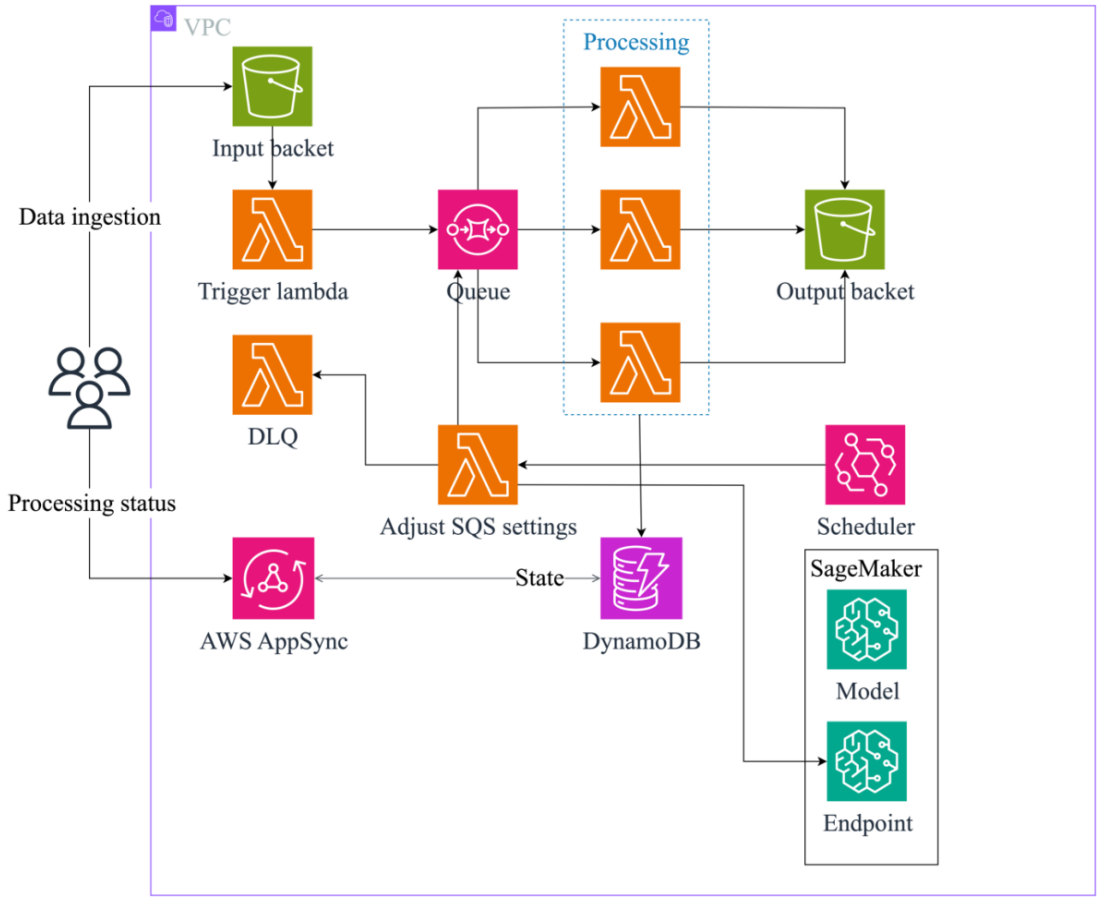

Рис. 2.5 – Архітектура фреймворку для розподіленої обробки даних

Опишемо детально основні рівні фреймворку.

1) Рівень завантаження даних.

На цьому рівні використовується механізм пакетної обробки даних та черга повідомлень. Повідомлення обробляються асинхронно, що забезпечує високу пропускну здатність з мінімальною затримкою. Цей рівень відповідає за ініціалізацію процесу обробки даних. При завантаженні файлів у Amazon S3 автоматично спрацьовує S3 trigger, який активує Lambda-функцію для попередньої обробки (розбиття даних на пакети). Пакети надсилаються у чергу повідомлень, що виступає буфером між рівнями.

Архітектуру рівня завантаження даних зображено на рис. 2.6.

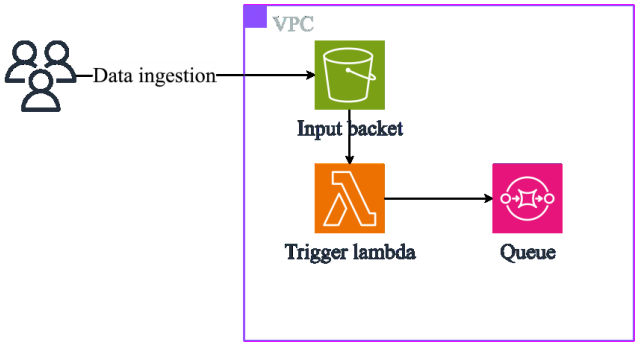

Рис. 2.6 – Архітектура рівня завантаження даних

2) Рівень обробки даних.

При появі нових повідомлень в черзі запускаються Lambda-функції. Головним завданням Lambda-функцій є обробка даних і збереження інформації про стан обробки. На цьому рівні відбувається безпосередня обробка кожного пакету даних. Основна логіка включає: читання повідомлення, обробку вмісту, збереження результату в S3, оновлення стану у DynamoDB.

Архітектуру рівня обробки даних та рівня управління стану зображено на рис. 2.7.

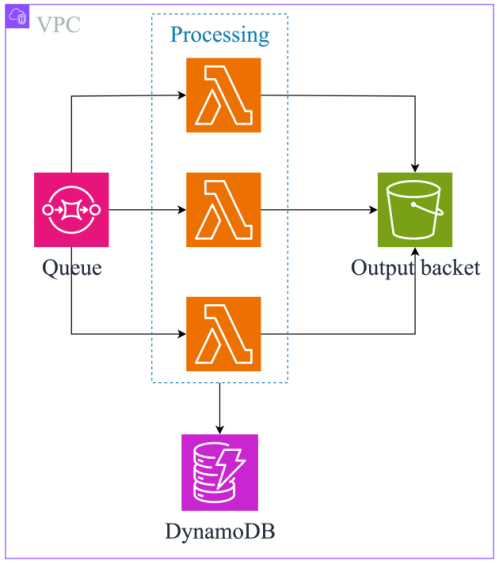

Рис. 2.7 – Архітектура рівня обробки даних та рівня управління стану

3) Рівень управління станом.

Для збереження стану обробки даних використовується DynamoDB. Наявність збереженого стану дає змогу проводити моніторинг виконання процесу обробки та відслідковувати помилки. Кожна оброблена частина даних має унікальний ідентифікатор і відповідну записану транзакцію, що відображає статус обробки.

4) Рівень моніторингу в реальному часі.

Для організації комунікації в реальному часі з підключеними клієнтами фреймворк застосовує AppSync. Такий підхід дозволяє отримувати інформації про хід розподіленої обробки даних практично без затримок. При зміні стану задачі у DynamoDB надсилається сповіщення клієнту у реальному часі (GraphQL Subscription), що дозволяє користувачу відслідковувати прогрес без перезавантаження інтерфейсу [34].

Архітектуру рівня моніторингу в реальному часі зображено на рис. 2.8.

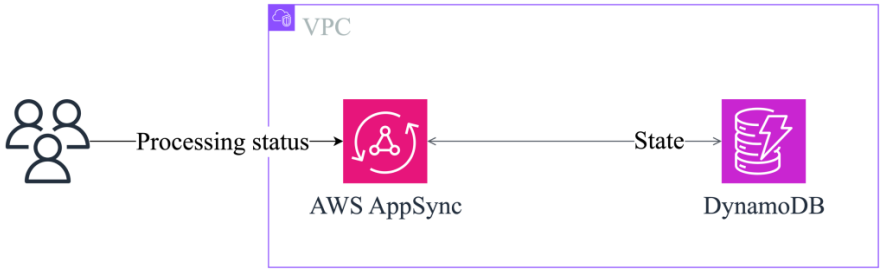

Рис. 2.8 – Архітектура рівня моніторингу в реальному часі

5) Рівень прогнозного автомасштабування.

Важливим елементом фреймворку є модуль прогнозного автомасштабування, який отримує інформацію про вхідні дані та оцінює ймовірність переходу системи між різними станами навантаження. На основі прогнозів здійснюється коригування основних параметрів Lambda та SQS з метою оптимізації використання ресурсів.

Архітектуру рівня прогнозного автомасштабування зображено на рис. 2.9.

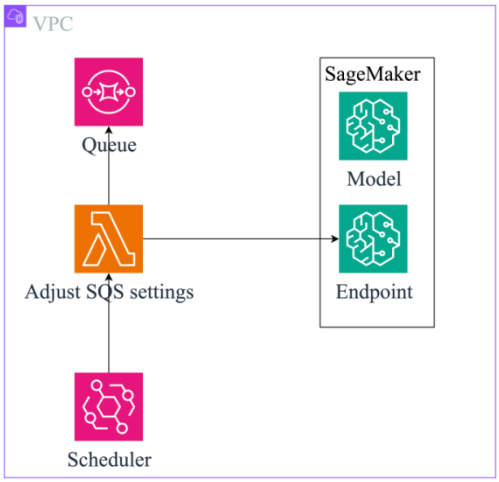

Рис. 2.9 – Архітектура рівня прогнозного автомасштабування

Основним функціональним призначенням модуля прогнозного автомаштабування є визначення ймовірності зміни режимів роботи системи під впливом навантаження та коригування основних параметрів черг і обробників даних [111]. Для передбачення навантаження використовуються напівмарковські моделі, розгорнуті в Amazon SageMaker, які є ефективними для моделювання динамічних подій, таких як зміни навантаження в розподілених безсерверних архітектурах. У порівнянні з традиційними Марковськими системами вони забезпечують більший реалізм, оскільки не обмежують тривалість перебування в станах лише експоненціальними розподілами. Така властивість є корисною для додатків зі змінним навантаженням, що сприяє точнішому передбаченню використання ресурсів і оцінці продуктивності. Крім того, напівмарковські процеси є ефективними для рішень з кількома режимами роботи [103, 112-114]. Моделі навчаються

на основі різних показників навантаження, таких як кількості вхідних подій, середнього часу обробки кожної події, довжини черги, кількості запущених функцій AWS Lambda. SageMaker використовує ці дані для побудови ймовірностей переходів та розподілів часу затримки в кожному стані.

Конфігураційні параметри SQS і Processing Lambda змінюються на основі прогнозу, отриманого з SageMaker у відповідності до наведеного нижче алгоритму [103]:

Algorithm 1. AWS Lambda and SQS parameters adjusting

```
1: function PredictiveAutoScaler(SMC_Model, P_INTERVAL, THRESHOLDS)
2:   loop every P_INTERVAL seconds
3:     metrics ← CollectMetrics ()
4:     sqs ← metrics.sqs_message_count
5:     λ ← metrics.lambda_concurrency
6:     current_state ← ClassifyState(sqs, THRESHOLDS)
7:     prediction ← QuerySageMaker(SMC_Model, current_state, metrics)
8:     next_state ← prediction.next_state
9:     if next_state ∈ {High_Load, Overload} then
10:        IncreaseLambdaConcurrency()
11:        IncreaseSQSPollingRate()
12:    else if next_state = Low_Load then
13:        DecreaseLambdaConcurrency()
14:        DecreaseSQSPollingRate()
15:    end if
16:    PushAppSyncUpdate(current_state, next_state, prediction.transition_time)
18:    Sleep(P_INTERVAL seconds)
19:  end loop
20: end function
```

де:

- SMC_Model – напівмарківська модель;
- P_INTERVAL – час, через який буде отриманий новий прогноз;

- THRESHOLDS – порогові значення для визначення стану.

На основі прогнозних даних з метою оптимізації використання обчислювальних ресурсів здійснюється коригування основних параметрів черг і обробників. Поєднання напівмарковських моделей з наведеним алгоритмом у вигляді модуля дозволяє прогнозувати сплески у робочому навантаженні та автоматично масштабувати обчислювальні ресурси.

Діаграма роботи фреймворку представлена на рис. 2.10.

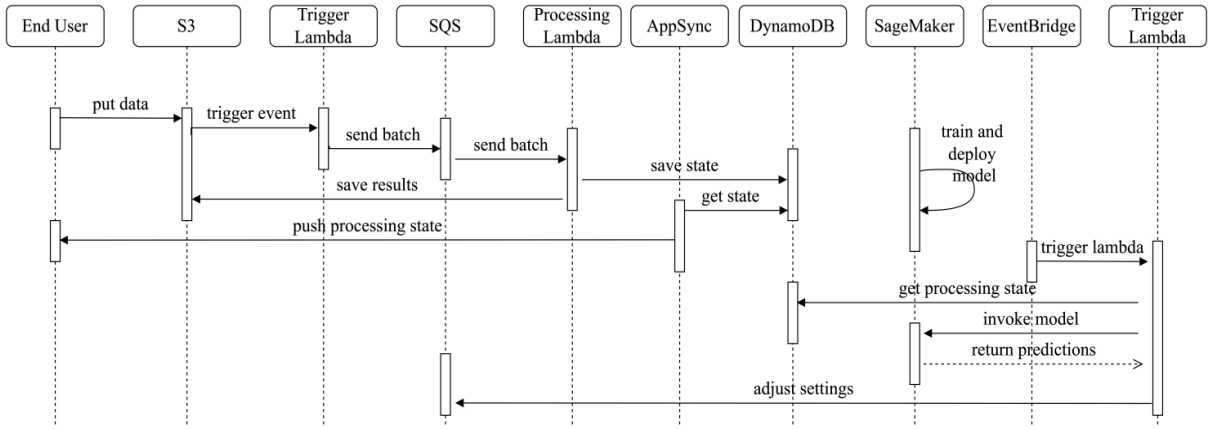

Рис. 2.10 – Діаграма роботи фреймворку

Вона охоплює такі основні етапи:

– користувач завантажує дані для обробки на S3;

– S3 активує Trigger Lambda;

– Trigger Lambda готує пакети завдань та надсилає повідомлення в чергу;

– черга запускає одну або декілька Processing Lambda та передає отримані повідомлення на вхід;

– Processing Lambda обробляє дані, зберігає результат в S3 та оновлює стан процесу обробки в DynamoDB;

– DynamoDB надсилає оновлення AppSync;

– AppSync транслює оновлення стану обробки користувачу;

– планувальник за розкладом запускає спеціальну функцію AWS Lambda, яка звертається до SageMaker Endpoint за прогнозом про майбутнє навантаження і оновлює конфігураційні параметри SQS і Processing Lambda.

Таким чином, розроблений фреймворк для розподіленої обробки даних

є основою інформаційної технології для оптимізації безсерверних обчислень із проактивним автоматичним масштабуванням обчислювальних ресурсів (рис. 2.11), практична реалізація та експериментальна апробація якої представлені в розділі 4.

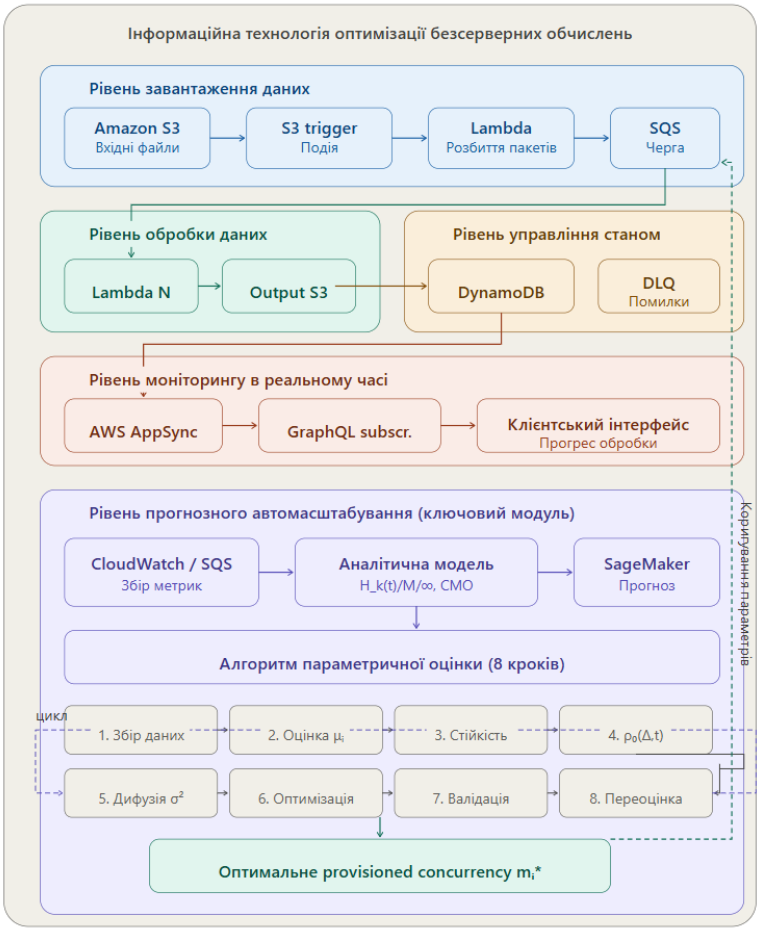

Рис. 2.11 – Архітектура інформаційної технології

Висновки до розділу 2

У даному розділі проведено аналіз архітектурних підходів до реалізації розподілених безсерверних обчислень у хмарних середовищах. Це дало змогу обґрунтувати вибір та розробити архітектуру власного фреймворку для оптимізації розподіленої обробки даних.

У результаті проведеного дослідження:

– Проаналізовано роль та значення подієво-орієнтованих архітектур для побудови безсерверних систем. Встановлено, що подійно-орієнтовані архітектури дозволяють забезпечувати

слабку зв’язаність між компонентами та масштабувати окремі функції незалежно, і є найбільш придатними для хмарних систем із динамічним навантаженням. Вони значно спрощують розробку та підтримку розподілених додатків. Модель "оплата за фактичне використання" та автоматичне масштабування ресурсів у відповідь на події роблять такі архітектури економічно ефективними, особливо для систем з нерівномірним навантаженням.

– Проведено порівняльний аналіз ключових шаблонів комунікації у безсерверних архітектурах, а саме: "Запит-Відповідь", "Публікація/Підписка", "Черга повідомлень" та "Шлюз API". Визначено їхні переваги та недоліки. Показано, що вибір шаблону значною мірою визначає продуктивність і масштабованість системи. Зокрема, для задач асинхронної розподіленої обробки даних найбільш доцільним є використання черг повідомлень, які забезпечують буферизацію та згладжування пікових навантажень, хоча і мають певні обмеження, такі як відсутність гарантії порядку доставки у стандартних чергах. Узагальнення цих шаблонів дозволило розробити критерії для обґрунтованого вибору архітектурного рішення під конкретну задачу.
– Проведено детальний порівняльний аналіз двох підходів до реалізації комунікації в реальному часі: на базі AWS AppSync та на базі WebSockets через API Gateway. Аналіз за критеріями простоти розгортання, масштабованості, безпеки, обробки стану та вартості показав, що AWS AppSync є кращим вибором для систем, що вимагають швидкої інтеграції з іншими сервісами AWS, вбудованої підтримки авторизації та автоматичного масштабування, тоді як API Gateway надає більше гнучкості, але потребує ручного налаштування багатьох аспектів.

– Зазначено обмеження реактивного масштабування, зокрема холодні старти, непередбачувані затримки та слабкий контроль з боку розробників. Встановлено, що стандартне реактивне масштабування, хоч і є економічно вигідним, призводить до значних затримок ("холодних стартів") при раптовому збільшенні навантаження. Це підтверджує актуальність досліджень, спрямованих на розробку проактивних методів масштабування, які здатні прогнозувати майбутнє навантаження та превентивно готувати необхідні ресурси.
– Обґрунтовано необхідність розробки нового фреймворку для розподіленої обробки даних, який би поєднував переваги безсерверної архітектури з інтелектуальним проактивним масштабуванням. Запропоновано гіпотезу, що використання прогнозного автомасштабування у поєднанні з чергами повідомлень дозволить значно зменшити час відповіді системи, мінімізувати кількість холодних запусків та підвищити загальну продуктивність і ефективність використання ресурсів.
– На основі проведеного аналізу запропоновано та описано багаторівневу архітектуру фреймворку, яка складається з рівня завантаження даних, рівня їх обробки, рівня управління станом, рівня моніторингу в реальному часі та рівня прогнозного автомасштабування. Така архітектура забезпечує необхідну ізоляцію та слабку зв'язність окремих компонент системи. Також сформульовано детальні функціональні та нефункціональні вимоги до фреймворку, що є основою для його подальшої реалізації та експериментальної перевірки.

Основні результати даного розділу опубліковані в роботах [33, 34, 103, 111]. Вони є основою для практичної реалізації та експериментального дослідження, що будуть представлені у розділі 3 та розділі 4.

РОЗДІЛ 3
ОЦІНЮВАННЯ ПАРАМЕТРІВ МОДЕЛЕЙ ЧЕРГ У БЕЗСЕРВЕРНИХ ОБЧИСЛЕННЯХ

3.1 Моделі черг та їх параметри

3.1.1 Безсерверні обчислення як система масового обслуговування (СМО)

Безсерверні обчислення є одним із найбільш перспективних напрямів розвитку хмарних технологій, що реалізують подійно-орієнтовану модель виконання, автоматичне масштабування ресурсів та модель оплати за фактичне використання. Базовим структурним елементом у безсерверних обчисленнях є AWS Lambda [115], яка широко застосовується для реалізації масштабованих розподілених застосунків.

З точки зору поведінки, безсерверні платформи доцільно розглядати як СМО, функціонування яких описується випадковими процесами. Такий підхід активно використовується у сучасних дослідженнях для опису процесів надходження завдань, їх обробки, перебування у черзі та автоматичного масштабування ресурсів.

У AWS Lambda вхідний потік формується як поєднання завдань з незалежних джерел подій, кожне з яких характеризується власною інтенсивністю та статистичними властивостями. До таких джерел належать HTTP-запити, події з об’єктних сховищ, черги повідомлень, шини подій та IoT-пристрої. У результаті сумарний потік завдань є неоднорідним і нестаціонарним.

Експериментальні дослідження показують, що навантаження у безсерверних системах часто має властивості самоподібності та сплесковості, що не дозволяє використовувати прості стаціонарні моделі і потребує застосування розширених стохастичних підходів [84, 116]. У роботах [117, 118] зазначено, що поєднання потоків подій є однією з основних причин складності аналітичного моделювання безсерверних архітектур.

Процес виконання завдань у AWS Lambda здійснюється шляхом автоматичного запуску ізольованих контейнерів з окремими екземплярами функції. Кількість таких контейнерів може змінюватись відповідно до поточного навантаження та, з теоретичної точки зору, може вважатися необмеженою. На практиці масштабування обмежується квотами хмарної платформи, однак ці обмеження є достатньо високими для більшості застосунків.

З точки зору теорії черг така система наближається до неоднорідної СМО, що дозволяє використовувати моделі $M/G/\infty$ або їх модифікації [117]. Водночас додатковим фактором впливу є наявність затримок при розгортанні та ініціалізації контейнерів, які мають випадковий характер і значно впливають на часові характеристики системи [119].

Час виконання функцій у AWS Lambda залежить від типу завдання та конфігурації ресурсів, зокрема обсягу виділеної пам’яті, який визначає доступну обчислювальну потужність CPU. Таким чином, інтенсивність обслуговування не є сталою величиною, а визначається параметрами конфігурації функції.

Низка робіт демонструють, що така модель відповідає СМО зі змінною швидкістю обслуговування та потребує адаптивних моделей, які враховують залежність часу обробки від виділених ресурсів [117, 118, 120].

AWS Lambda може бути викликана як в асинхронному, так і синхронному режимі. Асинхронні виклики передбачають наявність практично необмеженої черги очікування, тоді як синхронні виклики мають жорсткі обмеження на час відповіді, що формує обмежену чергу та допускає втрату завдань.

Вартість виконання функцій у AWS Lambda прямо пропорційна часу виконання та обсягу виділеної пам’яті. Це вводить додатковий економічний критерій оптимізації поряд із класичними показниками продуктивності. У роботах [118, 120] зазначається, що оптимізація безсерверних систем є багатокритеріальною задачею, яка поєднує часові та вартісні метрики.

Таким чином, безсерверна платформа може розглядатися як СМО зі змішаними режимами роботи в якій вхідний процес є поєднанням декількох потоків різної природи, а процес обслуговування характеризується змінною інтенсивністю, що залежить від конфігурації активних серверів, що підтверджується сучасними дослідженнями продуктивності [121].

3.1.2 Модель суміші потоків завдань у безсерверних обчисленнях

У теорії черг процес надходження завдань до СМО $M(t)/M/c$ традиційно описується як випадковий процес $N(t)$, який у класичному випадку моделюється неоднорідним пуассонівським процесом $M(t)$ з інтенсивністю $\lambda(t)$. У цьому випадку кількість завдань на інтервалі $[t_1,\ t_2]$ буде пропорційною середній інтенсивності завдань на даному інтервалі [153], тобто

$$E\big(N(t_2) - N(t_1)\big) = \int_{t_1}^{t_2} \lambda(t)dt. \qquad (3.1)$$

Фізичний та прикладний зміст даного співвідношення зрозумілий із врахуванням властивостей процесу Пуассона $N(t)$ для надходження заявок із змінною інтенсивністю $\lambda(t)$. Такий процес широко застосовується для моделювання вебнавантажень, потоків подій та систем із змінною інтенсивністю трафіку в часі, що підтверджується сучасними дослідженнями у сфері хмарних і безсерверних обчислень [122, 123].

Разом з тим, практика застосування безсерверних архітектур полягає у тому, що вхідний потік формується не одним джерелом, а множиною незалежних тригерів подій, які асинхронно ініціюють виконання функцій. Розглянемо випадок надходження завдань, що формується шляхом поєднання декількох незалежних потоків, кожен з яких відповідає окремому джерелу подій. Такі потоки інтерпретуються як випадкові процеси надходження завдань від різних сервісів.

Для безсерверних обчислень вхідний потік формується з k незалежних джерел (тригерів). Наприклад, для типової безсерверної архітектури на AWS:

- $N_1(t)$ – потік HTTP-запитів через API Gateway (рис. 3.1);

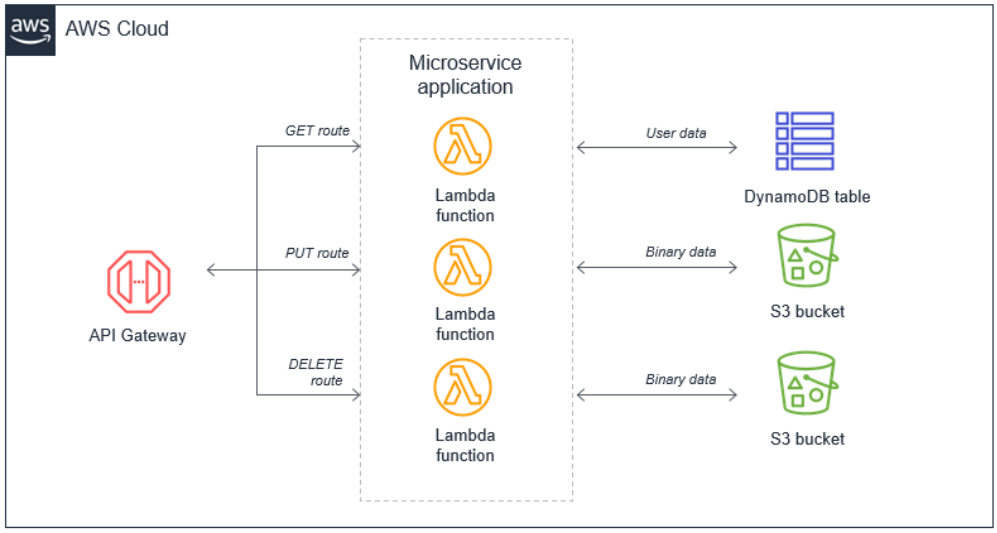

Рис. 3.1 – Безсерверна архітектура з використанням API Gateway і Lambda [124]

- $N_2(t)$ – потік подій від S3 (завантаження файлів, рис. 3.2);

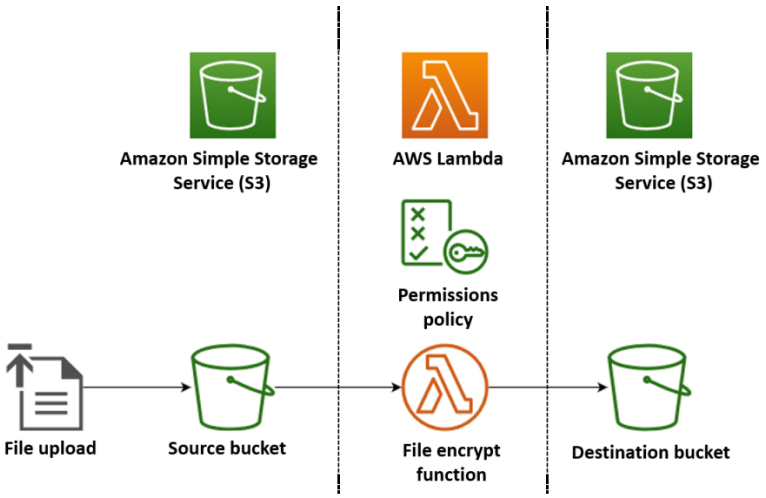

Рис. 3.2 – Безсерверна архітектура для обробки файлів [125]

- $N_3(t)$ – потік повідомлень з SQS черг (рис. 3.3);

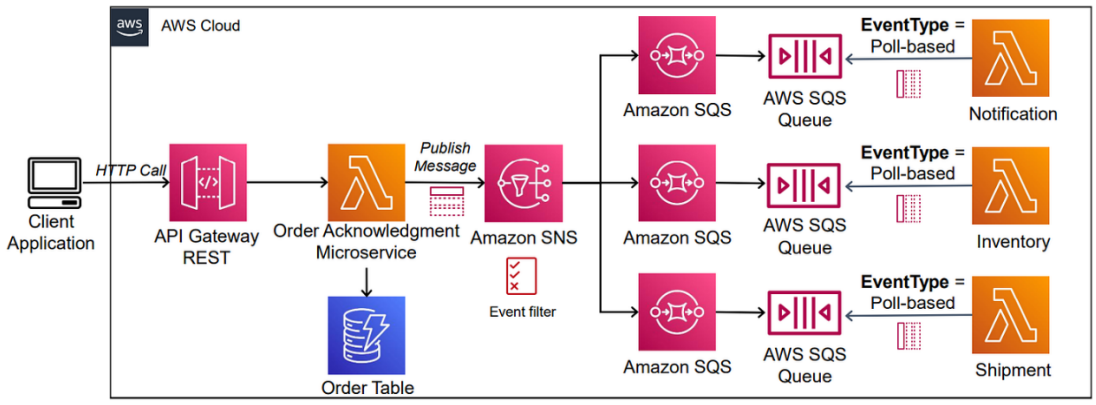

Рис. 3.3 – Безсерверна архітектура з використанням для Amazon SQS з AWS Lambda [126]

- $N_k(t)$ – потік подій від інших джерел (DynamoDB Streams, EventBridge тощо).

Наявність множини незалежних джерел подій у безсерверних системах підтверджується сучасними емпіричними дослідженнями робочих навантажень функцій, де показано, що виклики формуються різними типами тригерів (HTTP, черги подій, таймери) та мають суттєво різні статистичні характеристики [127, 128].

У цьому випадку [153] математичною моделлю процесу надходження завдань є суміш процесів надходження завдань $N_1(t)$, …, $N_k(t)$ із відповідними вагами $w_1(t), \dots, w_k(t)$, тобто

$$N(t) = \sum_{i=1}^{k} w_i(t) N_i(t), \tag{3.2}$$

де $w_i(t)$ визначає відсоток завдань, які надійшли за період часу $[t, t+h]$ при малому h, причому

$$\sum_{i=1}^{k} w_i(t) = 1\,, t \geq 0. \tag{3.3}$$

Використання вагових коефіцієнтів де $w_i(t)$ дозволяє:

- враховувати зміну структури навантаження у часі (наприклад, домінування SQS-подій у пакетних обробках і HTTP-трафіку у реальному часі) [129];

- будувати моделі прогнозування автоматичного масштабування для Lambda-функцій;
- оцінювати внесок кожного джерела у затримки та використання обчислювальних ресурсів.

Така модель відповідає сучасним підходам до опису композитних потоків у подійно-орієнтованих та мікросервісних архітектурах, де агрегований трафік розглядається як поєднання незалежних випадкових процесів з різними розподілами надходження завдань [130, 131].

Розглянуті вище підходи є дуже подібними, проте відображають концептуально протилежні прикладні задачі із різною природою надходження завдань. Різниця полягає саме у розмежуванні надходження завдань різної інтенсивності чи різної складності [153].

3.1.3 Пуассонівський процес з марковською модуляцією (MMPP)

Важливо зазначити принципову відмінність запропонованої моделі від класичного Markov-Modulated Poisson Process (MMPP) [132] – пуассонівського процесу з інтенсивністю, що змінюється під впливом латентного Марковського ланцюга станів. У MMPP процесі основним фактором є усереднена інтенсивність надходження завдань, що визначається наступним співвідношенням

$$\lambda(t) = \sum_{i=1}^{k} w_i(t)\lambda_i(t), \tag{3.4}$$

де $\lambda_i(t)$ – інтенсивність надходження з i-го каналу з ймовірністю надходження $w_i(t)$ у момент часу t [153]. Зрозуміло, що випадковий процес $N(t)$, визначений співвідношенням (3.2) та випадковий процес $\widetilde{N}(t)$ (3.4) має різні ймовірнісні характеристики, отже відповідні СМО, що описуються цими двома процесами за умов всіх інших характеристик, є різними. Даний факт, наприклад, можна показати, обчисливши дисперсії випадкових

процесів $N(t)$ та $\widetilde{N}(t)$ за умови попарної незалежності випадкових процесів $N_1(t), \dots, N_k(t)$:

$$Var(N(t)) = \sum_{i=1}^{k} \int_0^t w_i^2(s)\lambda_i(s)ds;$$

$$Var\left(\widetilde{N}(t)\right) = \lambda(t) = \sum_{i=1}^{k} \int_0^t w_i(s)\lambda_i(s)ds.$$

Оскільки $w_i(t) \in [0,1]$, завжди виконується $w_i^2(t) \leq w_i(t)$, отже:

$$Var(N(t)) \leq Var\left(\widetilde{N}(t)\right).$$

Ця різниця є суттєвою для безсерверних систем, оскільки дисперсія безпосередньо впливає на точність оцінок довжини черги та ймовірність перевищення лімітів одночасного виконання. MMPP-модель припускає, що всі джерела подій формують єдиний потік з усередненою інтенсивністю, тоді як модель (3.2) враховує те, що кожне джерело (API Gateway, S3, SQS) генерує незалежний потік із власними статистичними характеристиками. Саме тому модель (3.2) є більш адекватною для систем із різнорідними тригерами та статистичними властивостями, ніж класичний MMPP.

Таким чином, основні ймовірнісні характеристики основного випадкового процесу (3.2) відрізняються від MMPP процесу зі щільністю (3.4). Як буде показано нижче, дисперсія буде важливим фактором при аналізі граничних еволюцій в схемі дифузійної апроксимації [17].

3.1.4 Характеристики досліджуваної СМО

На основі введених означень досліджувана неоднорідна СМО описується наступним набором характеристик, що відображають специфіку безсерверних обчислювальних платформ.

Вхідний процес. Процес надходження завдань визначається процесом (3.2) та є сумішшю незалежних неоднорідних пуассонівських процесів з відомими параметрами $\left(w_i(t), \lambda_i(t)\right)$, $t \geq 0$. Такий підхід дозволяє явно врахувати гетерогенність джерел подій, що формують навантаження на

систему. У контексті AWS Lambda інтенсивності $\lambda_i(t)$ можуть бути оцінені на основі метрик Amazon CloudWatch, зокрема Invocations та ConcurrentExecutions, тоді як вагові коефіцієнти $w_i(t)$ визначаються розподілом трафіку між тригерами (API Gateway, S3, SQS, EventBridge тощо). Для джерел типу AWS SQS параметри вхідного процесу можуть бути безпосередньо оцінені на основі метрик NumberOfMessagesSent, яка визначає кількість повідомлень, надісланих до черги за інтервал часу, що відповідає інтенсивності надходження заявок $\lambda_i(t)$ для i-го джерела та NumberOfMessagesReceived, яка визначає кількість повідомлень, отриманих Lambda-функціями з черги, що відповідає інтенсивності обробки μ_i. Використання суміші процесів, а не одного процесу з усередненою інтенсивністю, узгоджується з сучасними роботами з моделювання безсерверних систем, де показано, що агрегування навантаження приховує істотну варіативність та призводить до систематичної помилки в оцінці дисперсії потоку завдань [31, 127, 133, 134].

Процес обробки. Процес обробки характеризується експоненціальним часом обробки із нескінченною (достатньо великою) кількістю серверів та інтенсивністю обробки $\mu_i,\ i \geq 0$. Така модель відповідає загальній практиці аналізу безсерверних платформ, де експоненціальний розподіл використовується як адекватна апроксимація часу виконання функцій при відсутності залежностей між викликами.

В AWS Lambda інтенсивність обробки μ_i залежить від:

- обсягу виділеної пам’яті,
- типу завдання (CPU-bound чи I/O-bound),
- ефектів холодного та теплого старту.

Будемо припускати, що обробка завдань на серверах є незалежною, тобто незалежним вважається час обробки завдань. Незалежність обробки завдань на серверах відповідає фактичній архітектурі Lambda, де кожен контейнер обробляє окремий виклик без спільного стану. Також припустимо, що кожен сервер буде характеризуватися вартістю c_i за одиницю часу та α_i –

усередненою кількістю відхилених завдань за одиницю часу за умови роботи відповідного сервера (throttling rate), що відображає обмеження на максимальну кількість одночасних завдань в обробці. Подібне поєднання часових і вартісних характеристик широко використовується в сучасних оптимізаційних моделях безсерверних обчислень [135, 136].

Кількість завдань. Кількість завдань в черзі системи являється необмеженою в Режимі 1 та обмеженою N_L в Режимі 2. Режим 1 відповідає асинхронним викликам:

- SQS - Lambda,
- EventBridge - Lambda,

де черга повідомлень може накопичувати заявки без явних обмежень. *Режим 2* відповідає синхронним викликам:

- API Gateway - Lambda,

де при перевищенні ліміту одночасного виконання запити відхиляються з помилкою 429 Too Many Requests.

Дане припущення дозволить оцінювати кількість відхилених завдань, а отже і кількість необроблених завдань. Врахування обмеженої черги є критичним для коректної оцінки SLA та ймовірностей втрати запитів, що підтверджується сучасними дослідженнями з аналізу QoS у безсерверних системах [137, 138].

Кількість процесів. Кількість процесів надходження завдань k на основі формули (3.2) є константою, тобто параметр k не залежить від часу. Для простоти будемо позначати через $A(t)$ – множину активних серверів в момент часу t, тобто тих серверів, які або обробляють деяке завдання, або готові (оплачені) для обробки завдання, через $A_W(t)$ – множину активних серверів в момент часу t, які обробляють деяке завдання в цей момент часу. Подібне формалізоване представлення динамічного набору серверів узгоджується з аналітичними моделями автоматичного масштабування в безсерверних середовищах [56, 139].

Інтенсивності надходження. Інтенсивності надходження $\lambda_i(t)$ обмежені функції, тобто $0 \leq \lambda_i(t) \leq \lambda_{MAX}$.

3.2 Оцінка параметрів СМО складної структури

Безсерверні обчислення як сучасна парадигма хмарних обчислень характеризуються автоматичним масштабуванням обчислювальних ресурсів, стохастичною природою часу виконання функцій та залежністю експлуатаційних витрат від поточного навантаження. Зважаючи на це, безсерверні платформи доцільно розглядати як СМО складної структури, для яких актуальними є методи асимптотичного аналізу, оптимізації параметрів та Марковські моделі.

Значна кількість наукових праць присвячена застосуванню центральної граничної теореми до аналізу СМО з великою або випадковою кількістю серверів, що є характерною ознакою безсерверних систем. Так, у роботі [140] розглянуто центральну граничну теорему для СМО, що складається з випадкової кількості серверів N з інтенсивностями μ_k, $k \geq 1$, причому основне припущення зроблене щодо незалежності цих серверів, тобто в основі роботи лежить одне з основних припущень центральної граничної теореми. Авторами використано наступне припущення щодо характеристик інтенсивностей надходження завдань:

$$\mu_k^N = \tilde{\mu}_k + \frac{1}{\sqrt{n}}\hat{\mu}_k, k \geq 1,$$

де пара $(\tilde{\mu}_k, \hat{\mu}_k)$, $\tilde{\mu}_k > 0$ повністю визначає асимптотичну поведінку інтенсивності k-го серверу при $N = n$. Для вхідного процесу визначено інтенсивність $\lambda^n >$ надходжень в залежності від кількості наявних серверів із наступною асимптотою, $\lim_{n\to\infty} \frac{\lambda^n}{n} = \lambda > 0$ та $\lim_{n\to\infty} \frac{\lambda^n - n\lambda}{\sqrt{n}} = \hat{\lambda}$. Кількість надходжень до моменту часу t визначена наступним співвідношенням $\sup\left\{l > 0 | \sum_{i=1}^{l} \frac{U_i}{\lambda^n} \leq t\right\}$, де U_i – строго додатні незалежні однаково

розподілені випадкові величини з $EU_i = 1$. Тоді, згідно з [140], кількість завдань у системі $X^n(t)$ при $n \to \infty$ має такий граничний процес:

$$X(t) = X_0 + \sigma W(t) + \beta t + \gamma \int_0^t \xi(s) ds,$$

де $W(t), t \geq 0$ – стандартний вінерівський процес, параметри σ, β, γ залежать від початкових параметрів процесу надходження та обробки завдань.

Подальший розвиток дифузійних апроксимацій для СМО представлено в роботі [141], де розглядається також центральна гранична теорема для систем масового обслуговування, причому робиться припущення, що СМО описується на основі ланцюга Маркова $Q^n(k)$, визначеного для інтенсивності надходження $\lambda_n = n$ та інтенсивності обробки завдань $\mu_n = a\left(n + \sqrt{n}\right)$. У цьому випадку нормалізований та центрований випадковий процес

$$Y^n(t) = X^n([(\lambda_n + \mu_n)t]),$$

де $X^n(k) = \frac{Q^n(k) - \lambda_n}{a\sqrt{n}}$, слабко збігається до вінерівського процесу $W(t)$, а точніше, має місце збіжність:

$$\lim_{n\to\infty} P(Y^n(t) \leq x | Y^n(0) = y) = \frac{1}{\sqrt{2\pi\sigma^2 t}} \int_{-\infty}^{x} e^{-\frac{1}{2\sigma^2 t}(z-y)^2} dz.$$

Ще однією роботою, спрямованою на дослідження асимптотичних властивостей СМО при великій кількості серверів та швидкому збільшенні інтенсивності надходження λ_n є робота [142], в якій проведено дослідження *G/G/m* СМО та їхніх асимптотичних характеристик. Одними з найбільш корисних результатів цієї роботи є наближення загальних СМО *G/G/m* на основі більш простих та вивчених *M/M/c* СМО, наприклад, зв'язок для оцінки часу очікування у двох системах. Одним із наближень в даній роботі є наближення часу очікування для *GI/M/m* моделі

$$P(W(GI/M/m; c_a^2) > 0) \approx$$

$$\approx \min\left\{1, \frac{1 - \Phi\left(2(1-\rho)\sqrt{m}(1 + c_a^2)^{-1}\right)}{1 - \Phi\left((1-\rho)\sqrt{m}\right)} P\left(W(M/M/m)\right)\right\},$$

де $c_a^2 = \frac{DX}{(EX)}$ – квадрат коефіцієнту варіації для випадкової величини X, що визначає надходження завдань за деякий фіксований інтервал часу (для експоненціального розподілу $c_a^2 = 1$). Також важливим результатом [142] є порівняльний аналіз різних СМО, причому автори розглядають також $H_2/M/m$ СМО, що є найпростішим процесом, побудованим на сумішах. Запропоновані апроксимації дозволяють здійснювати оцінку ймовірності очікування та часу затримки, що має безпосереднє практичне значення для аналізу продуктивності безсерверних обчислень.

Окрему групу робіт становлять дослідження, спрямовані на оцінку оптимальної кількості серверів у СМО з випадковими параметрами. У роботі [143] визначено простіший підхід для визначення оптимальної кількості серверів для СМО з випадковою інтенсивністю обробки завдань μ. У цьому випадку автори пропонують просте наближення кількості серверів $m \approx E\left(\frac{\lambda}{\mu}\right) + z_a \sigma_{\frac{\lambda}{\mu}}$, де $\sigma_{\frac{\lambda}{\mu}}$ – стандартне відхилення для випадкової величини $\frac{\lambda}{\mu}$, z_a – квантиль рівня a для стандартного нормального розподілу. У випадку деяких обмежень на розподіл випадкової величини $\frac{\lambda}{\mu}$ (наприклад, нормального розподілу) розглянуто поправку на оптимальну кількість серверів

$$m \approx E\left(\frac{\lambda}{\mu}\right) + z_a \sqrt{E\left(\frac{\lambda}{\mu}\right) + D\left(\frac{\lambda}{\mu}\right)}.$$

Такий підхід є релевантним для безсерверних систем, у яких час виконання функцій є випадковим процесом та істотно залежить від типу навантаження.

В роботі [144] розглянуто проблему визначення оптимальної кількості серверів для так званої нечіткої СМО, а саме розглянуто наступну СМО *M/M/s:K*, де K – максимально допустима кількість серверів, s – реальна кількість серверів в роботі (обробці завдань чи функцій). В даній роботі зроблено наступне припущення щодо інтенсивності надходження та обробки завдань

$$\lambda_n = \begin{cases} (K-n)\lambda, n = 0, \dots, K, \\ 0, n > K, \end{cases}$$

$$\mu_n = \begin{cases} n\mu, n = 1, \dots, s; \\ s\mu, n > s. \end{cases}$$

На відміну від попередніх робіт, дане припущення ґрунтується на оптимізаційній задачі, що мінімізує вартість обробки завдань, тобто

$$\mathrm{E}(CT) = E(CS) + E(CE) \to MIN,$$

де CS – середня загальна вартість за одиницю часу, CE – середня вартість обробки завдання за одиницю часу. Авторами роботи визначено знаходження ймовірностей переходу відповідного дискретного ланцюга Маркова з неперервним часом на основі наступного рівняння Колмогорова – Чепмена

$$p'(t) = A(\lambda, \mu)p(t),$$

де $A(\lambda, \mu)$ – генератор процесу Маркова, визначеного на основі інтенсивностей λ_n та μ_n, $p_i(t)$ – ймовірність того, що в час t система містить рівно i завдань та $\pi = \lim_{t\to\infty} p(t)$ стаціонарний розподіл системи. В роботі наведено алгоритм оцінки оптимальної кількості серверів на основі оптимізаційної задачі, визначеної вище, та стаціонарного розподілу π.

У роботі [145] розглянуто $M^X/M/m$ СМО при нескінченній кількості серверів $m = \infty$. Причому зроблено припущення, що дана СМО описується на основі Марковського процесу $J(t)$, $t \geq 0$ з генератором Q; кількість одночасних завдань визначається незалежними невід'ємними випадковими величинами X_i, що задовольняють умову $E\log(X_i + \varepsilon) < \infty$ для деякого $\varepsilon > 0$. Основним результатом роботи є той факт, що випадкова величина $N^{\frac{\beta}{2}}\left(\frac{M^{(N)}}{N} - \rho\right)$, де $\beta = \min(\alpha, 1)$, α – коефіцієнт масштабування для часу надходження, $M^{(N)}$ – нормована кількість завдань у черзі, збігається до нормального розподілу з нульовим середнім та дисперсією, залежною від параметрів СМО та коефіцієнта масштабування α. Використовуючи дане твердження, авторами запропоновано оцінку кількості серверів в СМО на основі довжини черги

$$P\left(M^{(N)} \le x\right) \approx \Phi_{(0,\sigma^2)}\left(N^{\frac{\beta}{2}}\left(\frac{x}{N} - \rho\right)\right),$$

де $\Phi_{(0,\sigma^2)}(x)$ – функція нормального розподілу з параметрами $(0, \sigma^2)$. Таким чином, на основі даного твердження можна здійснювати оцінку оптимальної кількості серверів для обмеженого процесу $M^{(N)}$ на скінченному інтервалі $[0, T]$.

У роботі [146] також розглянута схема оцінки оптимальної кількості серверів на основі нормального розподілу для замкнутої СМО з M кроками, причому i-й крок описується r_i паралельними серверами з інтенсивністю μ_i. У роботі показано, що сумісна ймовірність на кожному кроці $i = 1, \dots, M$ буде наближено підпорядковуватися нормальному розподілу, тобто

$$P(n_1, n_2, \dots, n_K) \approx$$

$$\approx \left(\prod_{i=1}^{K} \gamma_i(n_i, \alpha_i)\right)\left(\sum_{i=1}^{M} \sigma_i^2\right)\left(\sum_{i=K+1}^{M} \sigma_i^2\right)^{-1} e^{-(N_K - \bar{N}_K)^2\left(\sum_{i=K+1}^{M} \sigma_i^2\right)^{-1}},$$

де n_i – кількість необроблених завдань на i-му кроці, K – кількість початкових кроків, для яких здійснена оцінка, N_K – кількість необроблених завдань на цих кроках, σ_i^2 – дисперсія кількості необроблених завдань на i-му кроці. Таким чином, даний результат, дозволяє оцінювати оптимальну кількість серверів для багатокрокової СМО на основі попередньої оцінки та визначення загальної кількості серверів у СМО

$$P(L \le N) = \sum_{n_1 + n_2 + \dots + n_M \le N} P(n_1, n_2, \dots, n_K).$$

У роботі [147] розглянуто СМО з M серверами, причому ключовим припущенням даної роботи є той факт, що сервери можуть виходити з ладу та відправлятися на ремонт, причому також зроблено додаткове припущення про наявність деякої кількості серверів $M_a = L - M$, які знаходяться в резерві та одразу можуть замінювати сервери, які вийшли з ладу. Аналогічно попереднім роботам, в статті розглянуто неоднорідний ланцюг Маркова з

неперервним часом, який описує СМО з інтенсивностями надходження та обробки наступного вигляду

$$\lambda_n = \begin{cases} M\lambda + (S-n)\alpha, n = 0, \dots, S, \\ (L-n)\lambda, n = S+1, \dots, M+S = L, \\ 0, n > L, \end{cases}$$

$$\mu_n = \begin{cases} i\mu, n = i, \dots, M+S; \\ 0, n > M+S. \end{cases}$$

У роботі [148] розглянуто проблему мінімізації вартості виконання завдань за умов обмеженості черги у СМО, причому мінімізація вартості буде ґрунтуватися на мінімізації кількості серверів (резервних серверів S та активних серверів R). Загальна оптимізаційна задача має наступний вигляд:

$$T_{cost}(S,R) \to MIN_{S,R}$$

за умови, що

$$A.V. = \sum_{i=0}^{R}\sum_{j=0}^{S}\pi_{i,j} > A,$$

де $T_{cost}(S,R)$ – загальна вартість обслуговування (резервування) серверів за одиницю часу, π – стаціонарні ймовірності відповідного випадкового процесу, що описує СМО. Попередній аналіз проблеми показав, що цільова функція $T_{cost}(S,R)$ не є опуклою, а отже, поставлену задачу неможливо розв'язати на основі методів опуклого програмування. Також у даній роботі проведено аналіз трьох політик використання серверів та проведено аналіз оптимальних розв'язків для різних політик і визначено, що оптимальною є політика без затримки на ремонт сервера при наявній можливості ремонту. Автори в роботі [148] також використовують стаціонарні розподіли відповідного Марковського процесу для аналізу оптимальної кількості серверів для СМО *M/M/(S,s).* Ця система характеризується тим, що починаючи з S серверів одночасно із виконаним завданням систему залишає сервер, який це завдання виконував, поки кількість серверів не буде рівна s, після чого кількість серверів знову поновлюється до кількості S. У роботі описано твірний оператор Марковського процесу для процесів надходження

та обробки завдань із інтенсивностями λ та μ відповідно; показано що система володіє невироджених стаціонарним розподілом при

$$\lambda < \mu \frac{S-s}{\sum_{j=s+1}^{S} \frac{1}{j}},$$

а стаціонарний розподіл відповідного Марковського процесу визначений співвідношенням $\pi_{S+n} = \pi_S R^n$, де R – розв'язок допоміжного матричного рівняння другого порядку $R^2 A_2 + R A_1 + A_0 = 0$.

У роботі [149] проведено аналіз СМО, для якої основним параметром є процес вхідних завдань, що характеризується кусково-сталим з ймовірністю 1 випадковим процесом $A(t)$, причому в даному випадку на відміну від попередніх робіт $A(t)$ є напівмарковським процесом, визначеним на основі процесу Марковського відновлення (A_n, τ_n). Одним із основних припущень в даній роботі є припущення щодо усередненої динаміки $A(t)$ $\left(\lim_{t\to\infty} \frac{A(t)}{t} = \frac{\lambda}{a}\right)$, де $\lambda^{-1} = E(\tau_n - \tau_{n-1}), a^{-1} = E(A_n - A_{n-1})$. Основний результат роботи визначає слабку збіжність процесу кількості необроблених завдань в системі до вінерівського процесу $W(t)$ при відповідному центруванню та нормалізації при $T \to \infty$. Таким чином, на основі даної граничної теореми знову ж таки можна проводити оцінку оптимальної кількості серверів з використанням обмеження кількості необроблених заявок на протязі достатньо великого інтервалу $[0, T]$.

Наведені вище моделі добре узгоджуються з економічною природою безсерверних обчислень, у яких вартість безпосередньо залежить від кількості активних екземплярів функцій та часу їх функціонування. У цих роботах отримано нормальні та дифузійні апроксимації для процесів кількості завдань у системі, що дозволяє оцінювати ймовірність перевантаження та визначати параметри масштабування безсерверних систем на скінченних часових інтервалах.

Подальший розвиток моделей систем масового обслуговування складної структури представлено в роботах [150] та [151]. Зокрема, у роботі

[150] основну увагу приділено двокроковим СМО, для яких надходження завдань відбувається з інтенсивністю λ, а інтенсивність опрацювання завдань рівний μ_1 та μ_2 відповідно на першому та другому кроці, причому витрати на перебування в черзі рівні h_1 та h_2 для першого та другого кроку відповідно. Основна ідея полягає у використанні Марковських процесів прийняття рішень з множини допустимих політик П, де кожна політика $\pi \in$ П визначає як сервер для виконання кожного завдання, так і місце в черзі для кожного завдання. Робота фокусує свою увагу на мінімізації усередненої вартості перебування в черзі, що визначена наступним співвідношенням

$$g(\pi, x) = \lim_{n\to\infty} sup \frac{V_n(\pi, x)}{n},$$

де $V_n(\pi, x) = E_x[\sum_{k=0}^{n-1} c(X_k, d_k(X_k))]$ – усереднена вартість перебування в черзі для політики π для початкового значення x, де X_k – двовимірний вектор довжини черг у кроці k, d_k – розміщення завдань в чергах для k-го кроку, $c(x, d(x))$ – вартість перебування. В роботі визначено, що описана СМО володіє невиродженим стаціонарним розподілом, якщо $\lambda(\mu_1^{-1} + \mu_2^{-1}) < 2$; також показано, що оптимізаційна задача $g(\pi, x) \to MIN_{\pi\in\Pi}$ має принаймні один розв'язок, який є оптимальним для кожного кроку зокрема.

Ще один результат розподіленої СМО розглянуто в роботі [151], в якій зроблено припущення що вхідний процес визначено інтенсивністю λ, причому паралельно використовуються n серверів, кожен з яких містить m_i ядер зі швидкістю обробки s_i, і вхідна інтенсивність розподілена між сереверами пропорційно до λ_i, де $\lambda = \lambda_1 + \cdots + \lambda_n$. В роботі визначено, що оптимальними характеристиками подібної СМО є СМО із однаковими швидкостями обробки $s_1 = \ldots = s_n$ для idle-speed моделі та $s_i = \left(\frac{\lambda_i}{\beta m_i}\right)^{\alpha}$, де α – додатковий параметр для потужність процесору з швидкістю s, а саме s^{α}.

Таким чином, аналіз сучасних наукових досліджень свідчить, що методи асимптотичного аналізу, оптимізації параметрів систем масового обслуговування та Марковські моделі є теоретичною основою для побудови

адекватних математичних моделей безсерверних обчислень. Це, у свою чергу, створює передумови для розробки методів оцінки продуктивності, затримок та вартості функціонування безсерверних платформ у режимах автоматичного масштабування та масового навантаження.

3.2.1 Умови обмеженості черги

При проєктуванні безсерверних систем важливим є визначення умов, за яких черга не зростатиме до нескінченності, тобто система спроможна обробити всі вхідні запити без необмеженого зростання затримки або кількості відхилених запитів.

У роботі [152] автори пропонують наступне позначення СМО – $(M(t)/M/\infty)^k|M$ для випадку з нескінченною чергою та $(M(t)/M/L_{MAX})^k|M$ для СМО зі скінченною чергою L_{MAX}, причому остання літера означає незалежність процесів обробки завдань. В нашому випадку, враховуючи факт розгляду нормованого процесу Пуассона (3.2), будемо використовувати наступне позначення СМО – $H_k(t)/M/\infty$ або $H_k(t)/M/L_{MAX}$ для випадків скінченої та нескінченної черги. Структурну схему розглянутої СМО можна побачити на рис. 3.4.

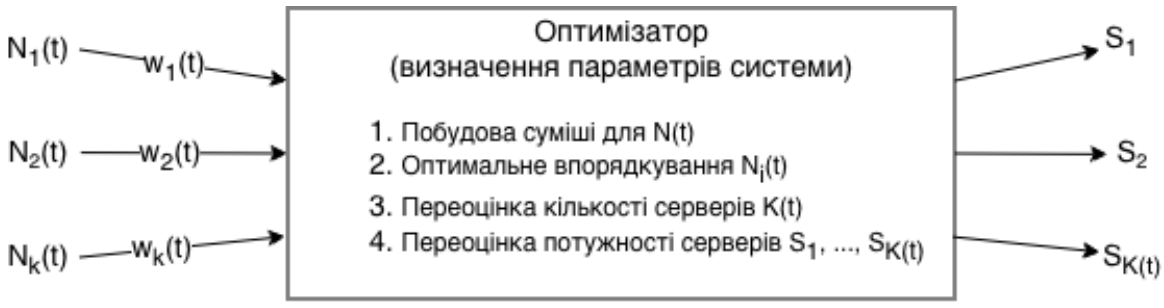

Рис. 3.4 – Структурна схема СМО для випадків скінченої та нескінченної черги

На вхід досліджуваної СМО надходять k незалежних потоків заявок $N_1(t)$, $N_2(t)$, …, $N_k(t)$ з відповідними вагами $w_1(t)$, $w_2(t)$, …, $w_k(t)$, що утворюють суміш процесів $N(t)$. Оптимізатор здійснює визначення

параметрів системи на основі вхідного потоку та поточного стану. На виході системи функціонують $K(t)$ серверів. Загалом оптимізатор реалізує чотири основні операції:

- побудова суміші для загального вхідного процесу

$$N(t) = \sum_{i=1}^{k} w_i(t) N_i(t);$$

- оптимальне впорядкування потоків $N_i(t)$ – визначення пріоритетів обробки для різних джерел подій відповідно до їхніх характеристик;
- переоцінка кількості серверів $K(t)$ для визначення оптимальної кількості функцій, що можуть миттєво обробляти запити (provisioned concurrency);
- переоцінка потужностей серверів $S_1, S_2, \ldots, S_{K(t)}$ – коригування конфігурації пам'яті Lambda-функцій, що визначає інтенсивності обробки.

Для початку розглянемо результат, який буде визначати умови зростання черги $L(t)$ до безмежності із часом для *Режиму 1*. Як було зазначено в класичних роботах, наявність стаціонарного режиму роботу СМО у випадку $\left(w_i(t), \lambda_i(t)\right)_{i=1}^{k} = (w_i, \lambda_i)_{i=1}^{k}$ або обмеженості черги $P(\forall t \geq 0 : L(t) < \infty) = 1$ визначаються виключно через інтенсивності процесу надходження та процесу оброки завдань. Для подальшого розгляду визначимо усереднені інтенсивності на скінченному інтервалі

$$0 \leq \Lambda(T) = \frac{1}{T} \int_0^T \sum_{i \in A(t)} w_i(t) \lambda_i(t)\, dt\,; \tag{3.5}$$

$$0 \leq \mathrm{M}(T) = \frac{1}{T} \int_0^T \sum_{i \in A(t)} \mu_i\, dt. \tag{3.6}$$

Використовуючи дані означення, визначимо умови обмеженості черги для $H_k(t)/M/\infty$ СМО [153].

Лема 3.2.1 [153]. Для того щоб черга $H_k(t)/M/\infty$ була обмеженою, необхідно та досить, щоб

$$\lim_{T\to\infty}\frac{\Lambda(T)}{\mathrm{M}(T)}<1. \tag{3.7}$$

Доведення. Спочатку відзначимо, що результат Леми 3.2.1 є зрозумілим інтуїтивно, оскільки усереднена інтенсивність надходження буде строго менша ніж усереднена інтенсивність обробки. Розглянемо випадковий процес

$$X(t)=N(t)-N_S(t),$$

де $N_S(t)$ – визначає процес обробок завдань, тобто

$$N_S(t)=\sum_{i\in A(t)}N_i(t),$$

причому $N_i(t), i\geq 1$ – незалежні випадкові процеси Пуассона із інтенсивностями μ_i. Зрозуміло, що розглянутий процес буде з ймовірністю 1 прямувати до $-\infty$, оскільки

$$EX(T)=E\big(N(T)\big)-E\big(N_S(T)\big)=T\big(\Lambda(T)-\mathrm{M}(T)\big)$$

та за умов незалежності процесів Пуассона

$$D\big(X(T)\big)=D\big(N(T)\big)+D\big(N_S(T)\big)=\sum_{i=1}^{k}\int_0^T w_i^2(s)\lambda_i(s)ds+\int_0^T\sum_{i\in A(t)}\mu_i\,ds.$$

Враховуючи той факт, що ваги $w_i(t)\in[0,1]$, одержимо

$$D\big(X(T)\big)\leq\sum_{i=1}^{k}\int_0^T w_i(s)\lambda_i(s)ds+\int_0^T\sum_{i\in A(t)}\mu_i\,ds=T\big(\Lambda(T)+\mathrm{M}(T)\big).$$

Використовуючи нерівність Чебишова, одержимо

$$P\left(X(T)>T\big(\Lambda(T)-\mathrm{M}(T)\big)+a\sqrt{T\big(\Lambda(T)+\mathrm{M}(T)\big)}\right)\leq$$

$$P\left(\big|X(T)-T\big(\Lambda(T)-\mathrm{M}(T)\big)\big|\geq a\sqrt{T\big(\Lambda(T)+\mathrm{M}(T)\big)}\right)\leq\frac{1}{a^2},a>0.$$

Враховуючи той факт, що $\Lambda(T)<\mathrm{M}(T)$ для достатньо великих T та підставляючи $a=\sqrt[4]{T}$, одержимо, що

$$T\big(\Lambda(T)-\mathrm{M}(T)\big)+\sqrt[4]{T}\sqrt{T\big(\Lambda(T)+\mathrm{M}(T)\big)}\to-\infty$$

з $P\left(X(T)>T\big(\Lambda(T)-\mathrm{M}(T)\big)+\sqrt[4]{T}\sqrt{T\big(\Lambda(T)+\mathrm{M}(T)\big)}\right)\le\frac{1}{\sqrt{T}}\to 0$ – з іншого.

Розглянемо тепер послідовність точок $t_i=Thi, h=\frac{1}{N}, i=1,\dots,N$ та $\Delta X(t_i)=X(t_i)-X(t_{i-1})-E\big(X(t_i)-X(t_{i-1})\big)$.

Використовуючи нерівність Ітемаді

$$P\left(\max_{i=1,\dots,N}\left|X(t_i)-E\big(X(t_i)\big)\right|\ge 3\sqrt[4]{T}\right)\le 3\max_{i=1,\dots,N}P\left(\left|X(t_i)-E\big(X(t_i)\big)\right|\ge\sqrt[4]{T}\right).$$

Проте $\max_{i=1,\dots,N}P\left(\left|X(t_i)-E\big(X(t_i)\big)\right|\ge\sqrt[4]{T}\right)\to 0$ для довільного N, що з врахуванням кусково-сталості $X(t)$, і доводить $P(X(t)\to-\infty)=1$.

Також слід відзначити, що

$$P(\forall t\ge 0: L(t)<\infty)=P(X(t)\to-\infty),$$

що і доводить твердження Леми 1.

Умова (3.7) визначає інтуїтивне правило, що безсерверна система є стабільною тоді і тільки тоді, коли середня пропускна здатність системи (визначена конфігурацією Lambda-контейнерів) перевищує середню інтенсивність вхідного потоку. У практичному плані це означає, що *reserved concurrency* та *provisioned concurrency* мають бути налаштовані так, щоб виконувалась умова $\frac{\Lambda(T)}{M(T)}<1$. Порушення цієї умови призводить до необмеженого зростання черги в AWS SQS або зростання кількості відхилених запитів [129].

3.2.2 Властивості поведінки черги

Для більш детального аналізу поведінки черги розглянемо випадковий процес, що описує кількість завдань у черзі, яку будемо позначати через $L(t)$ та кількість відмов, яку надалі будемо позначати через $\alpha(t)$. Зауважимо, що усереднена кількість відмов згідно наших припущень буде рівною

$$E\alpha(t)=\sum_{i\in A_W(t)}\alpha_i.$$

Перейдемо до більш детального розгляду динаміки випадкового процесу $L(t), t \geq 0$. Приріст даного процесу на інтервалі $[t, t+\Delta]$ для $\Delta \ll 1$ буде визначатися наступним співвідношенням

$$P(L(t+\Delta) - L(t) = m | L(t) = k) =$$

$$= \begin{cases} \sum\limits_{i \geq \max(m-k,0)} \frac{\left(\Delta\lambda(t)\right)^i e^{-\Delta\lambda(t)}}{i!} \frac{\left(\Delta\mu(t)\right)^{i+k-m} e^{-\Delta\mu(t)}}{(i+k-m)!}, m \geq -k+1, \\ \sum\limits_{i=0}^{\infty} \sum\limits_{j=i+k}^{\infty} \frac{\left(\Delta\lambda(t)\right)^i e^{-\Delta\lambda(t)}}{i!} \frac{\left(\Delta\mu(t)\right)^j e^{-\Delta\mu(t)}}{j!}, m = -k. \end{cases}$$

де $\mu(t) = \sum_{i \in A(t)} \mu_i$.

Розглянемо більш детально умовне математичне сподівання приросту, за умови $L(t) = k$:

$$E(L(t+\Delta) - L(t) | L(t) = k) = \sum_{m=-k}^{\infty} mP(L(t+\Delta) - L(t) = m | L(t) = k) =$$

$$-k * e^{-\Delta\left(\lambda(t)+\mu(t)\right)} \sum_{i=0}^{\infty} \sum_{j=i+k}^{\infty} \frac{\left(\Delta\lambda(t)\right)^i}{i!} \frac{\left(\Delta\mu(t)\right)^j}{j!} +$$

$$+e^{-\Delta\left(\lambda(t)+\mu(t)\right)} \sum_{m=-k+1}^{\infty} m \sum_{i \geq \max(m-k,0)} \frac{\left(\Delta\lambda(t)\right)^i}{i!} \frac{\left(\Delta\mu(t)\right)^{i+k-m}}{(i+k-m)!}.$$

Для $k = 0$ одержимо, що

$$E(L(t+\Delta) - L(t) | L(t) = 0) =$$

$$= \sum_{m=1}^{\infty} m \sum_{i=0}^{\infty} \frac{\left(\Delta\lambda(t)\right)^i e^{-\Delta\lambda(t)}}{i!} \frac{\left(\Delta\mu(t)\right)^{i-m} e^{-\Delta\mu(t)}}{(i-m)!} > 0.$$

Для достатньо великих $k > 0$ на основі означення ймовірностей для розподілу Пуассона при $\lambda(t) < \mu(t)$ одержимо, що

$$\phi(k, \Delta) = E(L(t+\Delta) - L(t) | L(t) = k) =$$

$$e^{-\Delta(\lambda(t)+\mu(t))}\left(\begin{array}{c} -k\sum_{i=0}^{\infty}\sum_{j=i+k}^{\infty}\frac{(\Delta\lambda(t))^i}{i!}\frac{(\Delta\mu(t))^j}{j!}+ \\ \sum_{m=-k+1}^{-1} m \sum_{i\geq\max(m-k,0)}\frac{(\Delta\lambda(t))^i}{i!}\frac{(\Delta\mu(t))^{i+k-m}}{(i+k-m)!}+ \\ \sum_{m=1}^{k-1} m \sum_{i\geq\max(m-k,0)}\frac{(\Delta\lambda(t))^i}{i!}\frac{(\Delta\mu(t))^{i+k-m}}{(i+k-m)!}+ \\ \sum_{m=k}^{\infty} m \sum_{i\geq\max(m-k,0)}\frac{(\Delta\lambda(t))^i}{i!}\frac{(\Delta\mu(t))^{i+k-m}}{(i+k-m)!} \end{array}\right).$$

Зрозуміло, що другий доданок більше за третій за модулем, а перший більше за четвертий за модулем, що означає $E(L(t+\Delta)-L(t)|L(t)=k)$ для достатньо великих k [153].

Лема 3.2.2 [153]. Нехай $0<\lambda(t)<\mu(t)$. Тоді умовне математичне сподівання $\phi(k)$ володіє наступними властивостями:

- $\phi(k,\Delta,t)>0$;
- $\phi(k,\Delta,t)$ – строго спадна функція по k;
- $\lim_{k\to\infty}\phi(k,\Delta,t)<0$ для довільного $\Delta>0, t\geq 0$.

Доведення. Перший факт випливає із означення функції $\phi(k)$. Для другого доданку різниця між

$$e^{\Delta(\lambda(t)+\mu(t))}\big(\phi(k,\Delta)-\phi(k-1,\Delta)\big)=$$

$$-k\sum_{i=0}^{\infty}\sum_{j=i+k}^{\infty}\frac{(\Delta\lambda(t))^i}{i!}\frac{(\Delta\mu(t))^j}{j!}+(k-1)\sum_{i=0}^{\infty}\sum_{j=i+k-1}^{\infty}\frac{(\Delta\lambda(t))^i}{i!}\frac{(\Delta\mu(t))^j}{j!}+$$

$$+\sum_{m=-k+1}^{\infty} m\sum_{i\geq\max(m-k,0)}\frac{(\Delta\lambda(t))^i}{i!}\frac{(\Delta\mu(t))^{i+k-m}}{(i+k-m)!}-$$

$$-\sum_{m=-k+2}^{-\infty} m\sum_{i\geq\max(m+1-k,0)}\frac{(\Delta\lambda(t))^i}{i!}\frac{(\Delta\mu(t))^{i+k-1-m}}{(i+k-1-m)!}=$$

$$=k(\Delta\mu(t))^{k-1}\sum_{i=0}^{\infty}\frac{(\Delta\lambda(t))^i}{i!}\left(\frac{(\Delta\mu(t))^i}{(i+k-1)!}-\frac{(\Delta\mu(t))^{i+k}}{(i+2k-1)!}\right)+$$

$$+\left(\sum_{i=0}^{\infty}\frac{\left(\Delta\lambda(t)\right)^{i}}{i!}\left(\frac{\left(\Delta\mu(t)\right)^{i+2k-1}}{(i+2k-1)!}-\sum_{j=i+k-1}^{\infty}\frac{\left(\Delta\mu(t)\right)^{j}}{j!}\right)\right)+$$

$$+\sum_{m=-k+2}^{-\infty}m\left(\sum_{i\geq\max(m-k,0)}\frac{\left(\Delta\lambda(t)\right)^{i}}{i!}\frac{\left(\Delta\mu(t)\right)^{i+k-m}}{(i+k-m)!}-\right.$$

$$\left.-\sum_{i\geq\max(m+1-k,0)}\frac{\left(\Delta\lambda(t)\right)^{i}}{i!}\frac{\left(\Delta\mu(t)\right)^{i+k-1-m}}{(i+k-1-m)!}\right).$$

Як ми можемо бачити, сума першого та другого доданку являється від'ємною за рахунок наявності в знаменниках $(i+2k-1)!$ та $(i+k-1)!$, з іншого боку – остання сума також є від'ємною, що і доводить друге твердження Леми 3.2.2.

Для доведення останнього твердження, зауважимо що при великих k, умовний розподіл випадкової величини $L(t+\Delta)-L(t)$ близький до розподілу $X(t+\Delta)-X(t)$, тобто

$$E\big(L(t+\Delta)-L(t)|L(t)\big)\approx E\big(X(t+\Delta)-X(t)\big)=$$

$$=\int_{t}^{t+\Delta}\big(\lambda(s)-\mu(s)\big)ds\approx\Delta\big(\lambda(t)-\mu(t)\big),$$

що і доводить той факт, що $\phi(k)<0$ для великих k, крім того

$$\lim_{k\to\infty}\phi(k,\Delta,t)=\lim_{k\to\infty}E\big(X(t+\Delta)-X(t)\big)=-\infty.$$

Лема 3.2.2 доведена.

Лема 3.2.2 показує, що зміна довжини черги Lambda-функції характеризується стабілізуючою поведінкою. Коли черга є малою, інтенсивність надходження запитів перевищує швидкість їх обробки, що зумовлює її зростання. Проте за великої довжини черги переважає процес обслуговування, унаслідок чого спостерігається спадна тенденція. Наявність точки рівноваги задає характерний (типовий) рівень навантаження системи та є визначальним орієнтиром для коректного налаштування механізмів автоматичного масштабування і параметрів provisioned concurrency.

3.2.3 Граничні еволюції у схемі усереднення та дифузійної апроксимації

Для більш детального аналізу поведінки черги розглянемо кусково-сталий напівмарковський процес $\tilde{L}(t) = L(t_n), t \in [t_n, t_{n+1})$, що описує кількість завдань у черзі та породжується процесом Марковського відновлення $\left(\tilde{L}_n = L(t_n), \tau_n = t_n = \frac{T}{N}n\right)$ [154]. Зауважимо, що на основі побудови, даний напівмарковський процес визначається напівмарковським ядром

$$Q(k, A, t) = P\left(\tilde{L}_n \in A | \tilde{L}_{n-1} = k, \tau_{n-1} = t\right) I_{\{t \le \tau_n < \Delta + t\}}. \quad (3.8)$$

Використовуючи даний процес можемо знайти граничне представлення для напівмарковського процесу $\tilde{L}(t)$, яке на основі властивості сепарабельності для $L(t)$, буде одночасно граничним представленням і для $L(t)$. Для цього будемо використовувати схему усереднення [18, 155] та схему дифузійної апроксимації [17, 156]. Для схеми усереднення визначимо сім'ю напімарковських процесів $\tilde{L}^{\varepsilon}(t), t \ge 0$ для $\varepsilon > 0$, що породжується процесом Марковського відновлення $\left(\tilde{L}_n^{\varepsilon} = \varepsilon L_n, \tilde{\tau}_n^{\varepsilon} = \varepsilon \tau_n\right)$. Для схеми дифузійної апроксимації визначимо визначимо сім'ю напімарковських процесів $\check{L}^{\varepsilon}(t), t \ge 0$ для $\varepsilon > 0$, що визначений наступною рівністю

$$\check{L}^{\varepsilon}(t) = \varepsilon L\left(\frac{t}{\varepsilon^2}\right) - \frac{\rho(t)}{\varepsilon} = \frac{1}{\varepsilon}\left(\varepsilon^2 \tilde{L}\left(\frac{t}{\varepsilon^2}\right) - \rho(t)\right),$$

де функція $\rho(t)$ буде визначена нижче, як розв'язок усередненого рівняння.

Для AWS Lambda дискретизація з кроком $\frac{T}{N}$ відповідає інтервалу збору метрик CloudWatch (типово 1 хвилина або 1 секунда), і стан системи $\tilde{L}_n$ описує кількість одночасних виконань у момент t_n.

Для формулювання основного твердження, аналогічно до роботи [17], введемо поняття компенсуючого оператора напівмарковського процесу, що задається процесом Марковського відновлення.

Означення 3.2.3. Компенсуючий оператор $\Gamma\phi(x, t), x \in R$ для напівмарковського випадкового процесу $\xi(t), t \ge 0$, що визначений

напівмарковським ядром $Q(x,A,t), x \in R, A \in \beta_R, t \geq 0$, наздається оператор

$$\Gamma\phi(x,t) = \frac{E\{\xi_n - \xi_{n-1} | \xi_{n-1} = x, \tau_n = t\}}{E\{\theta_n | \xi_{n-1} = x, \tau_{n-1} = t\}}. \quad (3.9)$$

Також позначимо інтенсивність часу перебування в стані через

$$b(x,t) = \frac{1}{E\{\theta_n | \xi_{n-1} = x, \tau_{n-1} = t\}}.$$

Тоді має місце наступне твердження.

Теорема 3.2.3 Мають місце наступні збіжності:

- У схемі усереднення випадковий процес $\tilde{L}^{\varepsilon}(t), t \in [0,T]$ слабко збігається при $\varepsilon \to +0$ до розв'язку диференціального рівняння

$$\frac{d\rho(t)}{dt} = \frac{\phi(\rho(t),\Delta,t)}{\Delta} \quad (3.10)$$

на скінченному інтервалі $[0,T]$ з початковою умовою $\rho(0) = L(0)$;

- У схемі дифузійної апроксимації випадковий процес $\check{L}^{\varepsilon}(t), t \in [0,T]$ слабко збігається при $\varepsilon \to +0$ до дифузійного процесу $\check{L}^{0}(t), t \in [0,T]$, що визначається стохастичним диференціальним рівнянням Ito

$$d\check{L}^{0}(t) = C_1\big(\check{L}^{0}(t), \rho(t), t\big)dt + \sqrt{\frac{1}{2}B(\rho(t),t)}dW(t) \quad (3.11)$$

на скінченному інтервалі $[0,T]$ з початковою умовою $\check{L}^{\varepsilon}(0) = \check{L}^{0}(0)$, де $C_1\big(\check{L}^{0}(t), \rho(t), t\big)$, $B(\rho(t),t)$ визначені в доведенні теореми.

Доведення. Для доведення умов збіжності у схемі усереднення, перевіримо необхідні умови на основі [18]. По-перше, стрибок $\tilde{L}_n - \tilde{L}_{n-1}$ має скінченний моменти другого порядку, оскільки на основі побудови випадкового процесу $L(t)$,

$$D\big(\tilde{L}_n - \tilde{L}_{n-1}\big) \leq D\big(X(t_n) - X(t_{n-1})\big) = \int_{t_{n-1}}^{t_n} \big(\lambda(s) + \mu(s)\big)ds < \infty;$$

Також, стрибок $\theta^n = \tau^n - \tau^{n-1} = \Delta$ задовольняє умову рівномірної інтегровності та обмеженості:

$$\sup_{k\geq 0} \int_T^{\infty} \bar{F}_k(t)dt = 0;$$

$$Ee^{-\varepsilon\theta^n} = e^{-\varepsilon\Delta} \leq 1 - \frac{\Delta}{2}\varepsilon, \varepsilon \ll 1.$$

Таким чином, твердження про збіжність в схемі усереднення є вірним.

Розглянемо тепер дослідження збіжності в схемі дифузійної апроксимації. Для цього зауважимо, що згідно побудови напівмарковського процесу $\check{L}^{\varepsilon}(t), t \geq 0$, одержимо наступні співвідношення для моментів

$$E\{\check{L}_n^{\varepsilon} | \check{L}_{n-1}^{\varepsilon} = x + \varepsilon x_1, \tau_{n-1} = t\} = \check{a}(x,t) + \varepsilon\check{a}_1(x,x_1,t) + o(\varepsilon);$$

$$E\left\{\left(\check{L}_n^{\varepsilon} - \check{a}(x,t) + \varepsilon\check{a}(x,x_1,t)\right)^2 | \check{L}_{n-1}^{\varepsilon} = x + \varepsilon x_1, \tau_{n-1} = t\right\} = \check{B}(x,t) + o(1);$$

$$E\{\check{L}_n^{\varepsilon} | \check{L}_{n-1}^{\varepsilon} = \rho(t) + \varepsilon x_1, \tau_{n-1} = t\} = E\{\check{L}_n^{\varepsilon} | \check{L}_{n-1}^{\varepsilon} = \rho(t), \tau_{n-1} = t\} + o(\varepsilon).$$

Третя рівність є наслідком збіжності в схемі усереднення та обмеженості моментів розподілів для процесу надходження та процесу обробки завдань. У цьому випадку одержимо, що всі умови збіжності в схемі дифузійної апроксимації [155] виконані, а отже має місце слабка збіжність $\tilde{L}^{\varepsilon}(t), t \geq 0$ до розв'язку стохастичного диференціального рівняння Ito

$$d\check{L}^0(t) = C_1\left(\check{L}^0(t), \rho(t), t\right)dt + \sqrt{\frac{1}{2}B(\rho(t),t)}dW(t),$$

де

$$C_1(x,\rho,t) = b(\rho,t)\check{a}_1(\rho,x,t),$$

$$B(\rho,t) = b(\rho,t)\left(\check{B}(\rho,t) - \check{a}(\rho,t)\right)^2.$$

Теорема 3.2.3 доведена.

Теорема 3.2.3 має важливе значення для аналізу безсерверних систем. Так, схема усереднення визначає детерміновану траєкторію $\rho(t)$ середнього навантаження Lambda-функції. Це рівняння відповідає закону великих чисел для процесу $\mathrm{L}(t)$ і дозволяє прогнозувати середню кількість одночасно

запущених Lambda-функцій на основі відомих параметрів $\left(w_i(t),\ \lambda_i(t)\right)$ та конфігурації серверів $A(t)$.

У свою чергу, схема дифузійної апроксимації визначає випадкові відхилення від середньої траєкторії. Граничний процес є процесом Орнштейна–Уленбека, що має важливу властивість: він є стаціонарним у широкому сенсі, тобто має обмежену дисперсію та кореляційну функцію, що експоненціально спадає [153]. Це відповідає центральній граничній теоремі для $L(t)$.

3.2.4 Оцінка довжини черги

На основі доведеної теореми, неоднорідний процес $L(t),\ t \geq 0$ підпорядковується закону великих чисел у схемі усереднення та центральній граничній теоремі у схемі дифузійної апроксимації [153]. Крім того, важливим є той факт, що граничний процес у схемі дифузійної апроксимації є процесом Орнштейна–Уленбека, а отже володіє деякими асимптотичними властивостями, які будуть визначальними у контексті апроксимації довжини черги в СМО $H_k(t)/M/\infty$. Одна із таких властивостей буде стосуватися наявності точки рівноваги для граничних процесів (3.10) та (3.11). Спочатку проаналізуємо усереднене рівняння (3.10), яке визначається на основі властивостей функції $\phi(k, \Delta, t)$. Згідно леми 3.2.2, усереднене рівняння (3.10) буде зростати для малих $\rho(t)$ та спадати при великих $\rho(t)$, що означає наявність точки рівноваги для кожного фіксованого t:

$$\rho_0(\Delta, t) \coloneqq \{x \in R_+ : \phi(x, \Delta, t) = 0\}. \qquad (3.12)$$

Функція $\rho_0(\Delta, t)$ є рівноважною функцією для розв'язку $\rho(t)$ і крім того близькість $\rho_0(\Delta, t)$ буде диктуватися співвідношенням $\frac{\Lambda(t)}{\mathrm{M}(t)}$, у випадку $\frac{\Lambda(t)}{\mathrm{M}(t)} \ll 1$, $\rho_0(\Delta, t)$ буде близьким до 0, що буде відповідати усередненій траєкторії випадкового процесу $L(t), t \geq 0$. Крім того, за виконання умови Леми 3.2.1, функція $\rho_0(\Delta, t)$ є обмеженою.

З іншого боку, процес Орнштейна–Уленбека, що задано стохастичним диференціальним рівнянням Ito (3.11), володіє тією ж рівноважною функцією $\rho_0(\Delta, t)$, тобто усереднений розв'язок (3.11) буде близьким в середньому до детермінованої функції $\rho_0(\Delta, t)$. Використовуючи дані міркування та задання випадкового процесу $\breve{L}^{\varepsilon}(t)$, можемо зазначити наступне наближення для випадкового процесу $L(t), t \geq 0$

$$L(t) = \frac{1}{\varepsilon}\breve{L}^{\varepsilon}(\varepsilon^2 t) + \frac{\rho(\varepsilon^2 t)}{\varepsilon^2} \approx \frac{1}{\varepsilon}\breve{L}^{0}(\varepsilon^2 t) + \frac{\rho(\varepsilon^2 t)}{\varepsilon^2},$$

де $\breve{L}^{0}(\varepsilon^2 t)$ визначено як розв'язок стохастичного диференціального рівняння Ito

$$\breve{L}^{0}(\varepsilon^2 t) = L(0) + \int_0^{\varepsilon^2 t} C_1\big(\breve{L}^{0}(s), \rho(s), s\big)ds + \frac{1}{2}\int_0^{\varepsilon^2 t} \sqrt{B(\rho(s), s)}dW(s)$$

або використовуючи міркування щодо $\rho_0(\Delta, t)$, одержимо

$$\breve{L}^{0}(\varepsilon^2 t) \approx L(0) + \int_0^{\varepsilon^2 t} C_1\big(\breve{L}^{0}(s), \rho_0(\Delta, s), s\big)ds + \int_0^{\varepsilon^2 t} \sqrt{\frac{1}{2}B(\rho_0(\Delta, s), s)}dW(s)$$

та

$$L(t) \approx$$

$$\frac{1}{\varepsilon}\left(L(0) + \int_0^{\varepsilon^2 t} C_1\big(\breve{L}^{0}(s), \rho_0(\Delta, t), s\big)ds + \int_0^{\varepsilon^2 t} \sqrt{\frac{1}{2}B(\rho_0(\Delta, s), s)}dW(s)\right) + \frac{\rho(\varepsilon^2 t)}{\varepsilon^2}.$$

Згідно даного наближення

$$L(t) \sim N\left(\frac{1}{\varepsilon}\left(L(0) + E\left(\int_0^{\varepsilon^2 t} C_1\big(\breve{L}^{0}(s), \rho_0(\Delta, t), s\big)ds\right)\right)\right.$$

$$\left. + \frac{\rho(\varepsilon^2 t)}{\varepsilon^2}, \frac{1}{2\varepsilon^2}\int_0^{\varepsilon^2 t} B(\rho_0(\Delta, s), s)ds\right),$$

або у більш спрощеній формі

$$L(t) \sim N\left(\rho_0(\Delta,t), \frac{1}{2\varepsilon^2}\int_0^{\varepsilon^2 t} B(\rho_0(\Delta,s),s)ds\right). \quad (3.13)$$

Таким чином, одержана проста оцінка для аналізу інтервалів надійності випадкового процесу $L(t), t \geq 0$, що характеризує довжину черги у неоднорідної СМО $H_k(t)/M/\infty$ [153]. Формула (3.12) є основним результатом для оцінки параметрів безсерверної системи. Вона стверджує, що розподіл кількості Lambda-функцій, які виконуються одночасно, $L(t)$ наближено підпорядковується нормальному розподілу з середнім $\rho_0(\Delta,t)$ та дисперсією, що визначається інтегралом від дифузійного коефіцієнта B.

Рівняння Ito для граничного процесу дозволяє оцінити ймовірність перевищення максимальної кількості функцій, які виконуються одночасно. Використовуючи нерівність Дуба, отримуємо оцінку:

$$P\left(\sup_{t\in[0,T]} L(t) < L_{MAX}\right), \quad (3.14)$$

де L_{MAX} – максимальна довжина черги, що відповідає ліміту одночасних виконань Lambda (reserved concurrency limit).

Для AWS Lambda ця оцінка має пряме практичне застосування:

- для L_{MAX} = reserved concurrency limit формула (3.14) визначає ймовірність того, що система не відхилятиме завдання протягом інтервалу *[0,T];*
- для L_{MAX} = account concurrency limit формула (3.13) оцінює ймовірність того, що інші Lambda-функції в акаунті не постраждають від конкурентного споживання ресурсів;
- на основі (3.13) та (3.14) можна визначити мінімальне L_{MAX} (тобто оптимальне значення reserved concurrency), що забезпечує задану ймовірність безвідмовної роботи.

3.3 Застосування паралельних та розподілених обчислень до СМО складної структури

3.3.1 Модель паралельної обробки в безсерверній архітектурі

У безсерверній архітектурі на основі AWS Lambda паралельна обробка є базовим механізмом масштабування. Кожен вхідний запит обробляється окремим контейнером, що відповідає моделі з нескінченною кількістю серверів ($G/M/\infty$) або великою, але обмеженою кількістю серверів. Дана модель природним чином узгоджується з концепцією event-driven архітектури, що є основою безсерверних обчислень [155, 157, 158].

Розглянемо конфігурацію з n групами Lambda-функцій, де i-та група має m_i зарезервованих контейнерів (provisioned concurrency) з інтенсивністю обробки μ_i, що визначається обсягом виділеної пам'яті та типом навантаження. Згідно з моделлю [151], вхідна інтенсивність розподіляється між групами: $\lambda = \lambda_1 + \cdots + \lambda_n$.

Для оптимізації продуктивності безсерверної системи необхідно визначити розподіл навантаження $(\lambda_1, \dots, \lambda_n)$ між групами функцій. Відповідно до результатів [151], оптимальною конфігурацією для мінімізації енергоспоживання є:

- однакова інтенсивність обробки $\mu_1 = \cdots = \mu_n$ для idle-моделі (всі Lambda-функції мають однакову конфігурацію пам'яті);
- $\mu_i = \left(\frac{\lambda_i}{\beta m_i}\right)^{\alpha}$ для моделі з різною потужністю, де α – параметр масштабування продуктивності відносно виділених ресурсів.

Для AWS Lambda параметр α відображає нелінійну залежність між обсягом виділеної пам'яті та продуктивністю CPU: AWS Lambda пропорційно масштабує CPU з пам'яттю, проте ефективність залежить від типу навантаження – CPU-bound чи I/O-bound. Дослідження [159] підтверджує, що вибір гранулярності функцій та конфігурації ресурсів може призводити до різниці до 53% у показниках SLO та до 54% у вартості ресурсів, що обґрунтовує необхідність математичної оптимізації.

3.3.2 Оцінка оптимальної кількості серверів

На основі результатів Теореми 3.2.3 та формули (3.13) пропонується метод оцінки оптимальної конфігурації безсерверної системи. Подібні задачі оптимізації для класичних СМО розглядались у роботах [143, 144, 148], проте для неоднорідних систем зі змішаними режимами роботи аналогічні результати відсутні.

Необхідно визначити кількість зарезервованих контейнерів $\{m_i\}_{i=1}^{n}$ для n груп Lambda-функцій із заданими параметрами обробки $(\mu_i, c_i^{\text{mem}})$, що мінімізує загальну вартість обслуговування при заданому рівні якості обслуговування.

Загальна вартість утримання серверів за одиницю часу складається з вартості provisioned concurrency для кожної групи контейнерів:

$$C_{\text{total}} = \sum_{i=1}^{n} c_i \cdot m_i \to \min,$$

де c_i – вартість утримання одного контейнера i-ої групи за одиницю часу. У термінах AWS Lambda вартість provisioned concurrency визначається як:

$$c_i = p_{\text{prov}} \cdot c_i^{\text{mem}},$$

де p_{prov} – вартість provisioned concurrency (USD за GB/s), c_i^{mem} – обсяг виділеної пам'яті i-го контейнера (GB). Додатково, вартість виконання одного запиту i-им контейнером визначається як

$$c_i^{\text{req}} = p_{\text{GB-s}} \cdot c_i^{\text{mem}} \cdot \mu_i^{-1} + p_{\text{req}},$$

де $p_{\text{GB-s}}$ – вартість обчислень (USD за GB/s), μ_i^{-1} – середній час обробки одного запиту (s), p_{req} – вартість одного запиту.

Оптимізація C_{total} здійснюється за умов:

$$P\left(\sup_{t\in[0,T]} L(t) < L_{\text{MAX}}\right) \geq 1 - \delta, \tag{3.15}$$

$$\lim_{T\to\infty} \frac{\Lambda(T)}{M(T)} < 1, \tag{3.16}$$

де δ – допустима ймовірність порушення обмеження (throttling probability), L_{MAX} – максимальна допустима кількість одночасних виконань (reserved

concurrency limit) або ліміт одночасних виконань на рівні облікового запису (account concurrency limit) в AWS Lambda).

Обмеження (3.15) відповідає вимозі Service Level Agreement (SLA) до відсотка успішно оброблених запитів [160, 161], а обмеження (3.16) гарантує стабільність системи згідно з Лемою 3.2.1.

Для оцінки ймовірності супремуму в (3.15) використаємо нерівність Дуба. Оскільки граничний процес $\check{L}^0(t)$ є розв'язком стохастичного диференціального рівняння Ito (3.10) і за умови, що дрейф $C_1\left(\check{L}^0(t), \rho(t), t\right)$ забезпечує обмеженість моменту $E\left|\check{L}^0(t)\right|^2$, $\check{L}^0(t)$ є мартингалом. Застосовуючи нерівність Дуба для субмартингалів [154]:

$$P\left(\sup_{t\in[0,T]} \check{L}^0(t) \geq a\right) \leq \frac{E\left|\check{L}^0(T)\right|^2}{a^2}, \quad a > 0.$$

З урахуванням зв'язку між $L(t)$ та $\check{L}^0(t)$, а також оцінки дисперсії $\check{L}^0(t)$ одержуємо достатню умову виконання (3.15) для визначення параметрів системи, тобто кількість та потужність серверів виконання:

$$\delta \approx \frac{E\left|\check{L}^0(T)\right|^2}{{L_{\text{MAX}}}^2}.$$

3.3.3 Оцінка кількості відхилених завдань

Для Режиму 2 (обмежена черга, $H_k(t)/M/L_{\text{MAX}}$) важливою характеристикою якості обслуговування є кількість відхилених завдань. У термінах AWS Lambda відхилені завдання – це запити, що отримали відповідь з кодом помилки 429 (Too Many Requests) через перевищення ліміту одночасних виконань (throttled invocations) [162, 163].

Визначимо інтенсивність відхилення як частку вхідного потоку, що перевищує пропускну здатність системи у момент часу t. За умови $L(t) > L_{\text{MAX}}$, усі нові запити, що надходять понад ліміт, відхиляються, що є коректним у випадку AWS Lambda. Тоді очікувана кількість відхилених запитів на інтервалі $[0, T]$ визначається як

$$R(T) = E\left(\int_0^T \mathbb{1}_{\{L(t)>L_{\text{MAX}}\}}\,\lambda(t)\,dt\right), \tag{3.17}$$

де $\lambda(t) = \sum_{i=1}^{k} w_i\,(t)\lambda_i(t)$ – загальна інтенсивність надходження заявок.

На основі нормального наближення (3.12), ймовірність перевищення ліміту в момент часу t оцінюється як

$$P(L(t) > L_{\text{MAX}}) \approx 1 - \Phi\left(\frac{L_{\text{MAX}} - \rho_0(\Delta, t)}{\sqrt{Var(L(t))}}\right), \tag{3.18}$$

де Φ – функція розподілу стандартного нормального розподілу, $\rho_0(\Delta, t)$ – рівноважна функція (3.12).

Підставляючи (3.18) у (3.17) та використовуючи теорему Фубіні для зміни порядку інтегрування та математичного сподівання, одержуємо наближення для очікуваної кількості відхилених запитів:

$$R(T) \approx \int_0^T \lambda\,(t)\left[1 - \Phi\left(\frac{L_{\text{MAX}} - \rho_0(\Delta,t)}{\sqrt{Var(L(t))}}\right)\right] dt. \tag{3.19}$$

Відповідно, частка відхилених запитів (throttling rate) на інтервалі $[0, T]$:

$$\theta(T) = \frac{R(T)}{\int_0^T \lambda(t)\,dt} \approx \frac{\int_0^T \lambda(t)\left[1-\Phi\left(\frac{L_{\text{MAX}}-\rho_0(\Delta,t)}{\sqrt{Var(L(t))}}\right)\right]dt}{\int_0^T \lambda(t)\,dt}. \tag{3.20}$$

Формула (3.20) має пряме практичне застосування: вимога SLA на рівні $\theta(T) \le \theta_{\text{SLA}}$ (наприклад, $\theta_{\text{SLA}} = 0{,}001$ для SLA 99,9%) може бути використана як альтернативне обмеження замість (3.15) у задачі оптимізації кількості зарезервованих контейнерів. Зауважимо, що підхід на основі прогнозованого показника відхилення запитів (3.20) є менш консервативним, ніж обмеження на супремум (3.15), оскільки він допускає короткочасні перевищення L_{MAX}, контролюючи лише їхню сумарну частку.

Подібний підхід до оцінки кількості відхилених заявок на основі нормального наближення використовувався у роботі [145] для $M^X/M/\infty$ СМО з Марковською модуляцією, де автори оцінювали $P\left(M^{(N)} \le x\right)$ на основі центральної граничної теореми. Результат (3.20) розвиває цю ідею на випадок неоднорідного вхідного процесу зі змішаними режимами роботи та неперервним часом.

3.3.4 Оцінка параметрів суміші та оптимальний розподіл завдань між серверами

Як було зазначено вище, вхідний процес можна представити у вигляді суміші розподілів, тобто має місце наступне представлення для вхідного процесу:

$$N(t) = \sum_{i=1}^{k} w_i(t) N_i(t),$$

де $w_i(t)$ визначає відсоток завдань, які надійшли за період часу $[t, t+h]$ при малому h [153], причому

$$\sum_{i=1}^{k} w_i(t) = 1\,, t \geq 0.$$

Одним із основних питань дослідження залишається оцінка невідомих параметрів вхідного процесу $(w_i(t), \lambda_i(t)),\ \ i = 1, \dots, k$. Знаходження оцінок даних параметрів можна здійснювати на основі методу оптимальної правдоподібності із застосуванням EM-алгоритму [164], що дозволяє швидко переобчислювати значення параметрів на інтервалі $[0, t+h]$ володіючи оцінкою параметрів на інтервалі $[0, t]$. Другим важливим аспектом оцінки є джерело оцінки, тобто на основі чого буде здійснюватися безпосередня оцінка. Зрозуміло, що на основі наявного процесу $N(t)$ неможливо зробити оцінку щодо кількості груп (кластерів) k, оскільки до процесу обробки завдань всі вхідні завдання є однотипними.

Для розв'язання цієї проблеми найефективнішим підходом є оцінка невідомих параметрів $(w_i(t), \lambda_i(t)),\ \ i = 1, \dots, k$ та безпосередньо параметра k на основі часу обробки завдань, що є найбільш інформативним джерелом для оцінки невідомих параметрів та, як показують приклади нижче, дозволяє достатньо швидко реагувати на зміну конфігурації самої системи. У контексті безсерверних обчислень джерелом даних для оцінки параметрів суміші є метрики AWS CloudWatch, зокрема Duration (час виконання Lambda-функції) та метрики SQS NumberOfMessagesReceived (кількість

оброблених повідомлень [103]), що дозволяє здійснювати оцінку параметрів $(w_i(t), \lambda_i(t)),\ i = 1, \dots, k$ безпосередньо з інфраструктурних метрик хмарної платформи. На рис. 3.5 можна побачити оцінку тривалості обробки вхідних завдань серверами. Даний підхід дозволяє виділити основні кластери вхідних потоків, а отже, оптимальним чином розрахувати необхідні потужності серверів для обробки завдань. Більш детальну інформацію про суміші пуассонівських розподілів можна знайти в роботі [165], причому в роботі є опис алгоритмів оцінки параметрів відповідних сумішей.

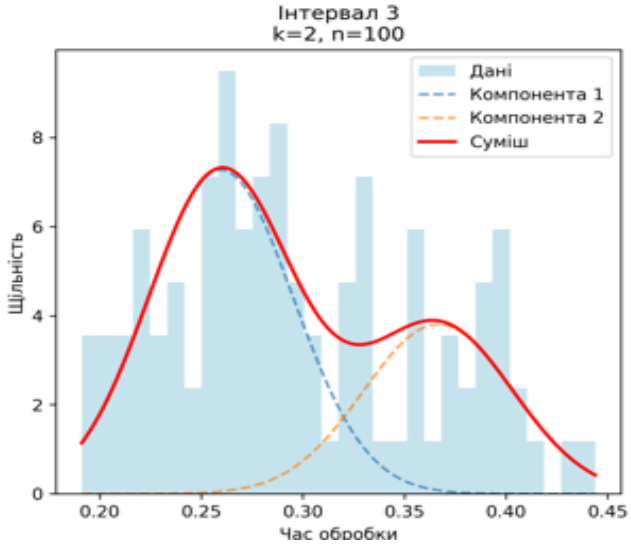

Рис. 3.5 – Оцінка параметрів суміші при k=2, де два кластери відповідають двом типам завдань з різним часом обробки, кожен з яких потребує різної конфігурації Lambda-функцій

Одразу слід зазначити, що основна увага зосереджена саме на класичній моделі вхідних параметрів, а саме робиться припущення про те, що вхідний процес є або процесом Пуассона, або сумішшю процесів Пуассона. Дане припущення значно спрощує дослідження відповідної системи з одного боку, а з іншого – дозволяє набагато простіше визначати граничні процеси у схемі усереднення та дифузійної апроксимації. Недоліком даного припущення є властивості процесу Пуассона, а саме незалежність приростів, що в нашому конкретному випадку означає незалежність завдань, які надходять із одного джерела. Найімовірніше, що вхідний процес матиме складнішу структуру ніж процес Пуассона, проте класичні припущення щодо

незалежності приростів дозволяють здійснювати точні наближення реальних процесів.

Таким чином, одержано практичний підхід (алгоритм) для визначення суміші розподілу та аналізу вхідного потоку завдань на основі наявних серверів або у випадку хмарних обчислень на основі визначення оптимальної кількості ресурсів, необхідних для найбільш швидкого обчислення, що в свою чергу повинно задовольняти прийнятний відсоток відхилених завдань.

Розглянемо ще один аспект оптимізації хмарних обчислень, який ґрунтуватиметься на основі перерозподілу вхідних завдань між серверами. Для подальшого викладу матеріалу, нагадаємо, що в момент часу t в системі налічується $K(t)$ серверів, причому припускатимемо, що швидкості обробки серверів розміщені в порядку спадання, тобто

$$\mu_1, \mu_2, \dots, \mu_{K(t)}.$$

В загальному випадку найпоширенішими є два підходи до розподілу завдань між серверами (протоколи виконання завдань), враховуючи швидкість обробки серверів:

- ***Випадковий вибір серверів***. За даного протоколу сервер для виконання певного завдання вибирається випадковим чином із вільних серверів, незважаючи на пріоритетність завдань або на можливе джерело завдань.
- ***Оптимальний вибір серверів***. У цьому випадку розглядається алгоритм, який певним чином оптимізує сумарний час виконання завдань, враховуючи специфіку самих завдань та, можливо, наявні проблематичні потоки завдань із великою ймовірністю відмови (неможливості виконання завдання). Найбільш розповсюджений підхід для оптимізації виконання завдань полягає у виборі найпотужнішого вільного сервера з наявних у цей час t вільних серверів. Інтуїтивно оптимальність даного підходу є зрозумілою, оскільки вибір серверів із максимальною швидкістю виконання означає, що «повільні» сервери

будуть обробляти менше завдань, а отже, загальна швидкість виконання буде значно вищою.

Для ілюстрації оптимальності підходу, розглянемо приклад *M/M/2* СМО з інтенсивністю надходження завдань λ та інтенсивностями виконання μ_1 та μ_2, причому для забезпечення стійкості роботи системи (існування нетривіального стаціонарного розподілу), припустимо, що

$$\frac{\lambda}{\mu_1 + \mu_2} < 1.$$

У цьому випадку при випадковому виборі сервера для виконання (вибирається довільний сервер з ймовірністю $\frac{1}{2}$) основні характеристики системи будуть наступними [166]:

- Ймовірність того, що СМО порожня дорівнює:

$$P_0 = \left(1 + \frac{\lambda(\mu_1 + \mu_2) + \lambda^2}{2\mu_1\mu_2} + \frac{\mu_1 + \mu_2}{2\mu_1\mu_2 - \lambda}\right)^{-1}.$$

- Середня кількість невиконаних завдань у черзі:

$$L_q = \frac{\lambda^3}{2\mu_1\mu_2(\mu_1 + \mu_2)(\mu_1\mu_2 - \lambda)} P_0.$$

- Середній час перебування в черзі:

$$W_q = \frac{\lambda^2}{2\mu_1\mu_2(\mu_1 + \mu_2)(\mu_1\mu_2 - \lambda)} P_0.$$

У випадку використання стратегії вибору найбільш потужного вільного сервера, дані характеристики приймуть наступні значення [166]

$$P_0^{opt} = \left(1 + \frac{\lambda}{\mu_1} + \frac{\lambda^2}{\mu_1(\mu_1 + \mu_2 - \lambda)}\right)^{-1};$$

$$L_q^{opt} = \frac{\lambda^3}{\mu_1(\mu_1 + \mu_2 - \lambda)^2} P_0;$$

$$W_q^{opt} = \frac{\lambda^2}{\mu_1(\mu_1 + \mu_2 - \lambda)^2} P_0.$$

За умови $\mu_1 > \mu_2$ неважко показати, що

$$P_0^{opt} > P_0$$

та

$$L_q^{opt} < L_q, W_q^{opt} < W_q.$$

Аналогічний результат одержано при роботі із багатьма серверами. Зокрема, імітаційне моделювання методом Монте-Карло для системи з трьома серверами ($\mu_1 = 0{,}03, \mu_2 = 0{,}06, \mu_3 = 0{,}09$ задач/с) та обмеженою чергою $N_L = 10$ при 100 прогонах підтвердило, що оптимальний протокол забезпечує зменшення частки відхилених завдань з 0,0193 до 0,0190 (покращення 1,45%, $p = 0{,}0474$) та зменшення середньої довжини черги з 3,2960 до 3,2862.

Проблематика підходу до вибору сервера з максимальним середнім часом виконання полягає в тому, що основне навантаження буде зосереджено на «швидких» серверах. З іншого ж боку, «повільні» сервери при невеликій кількості завдань будуть більше перебувати в режимі очікування, що у випадку використання хмарних технологій призведе до втрати коштів через оплату ресурсів, які фактично не працюють. В AWS Lambda дана проблема безпосередньо пов'язана з вартістю provisioned concurrency, де оплата за прогріті контейнери здійснюється за одиницю часу незалежно від того, чи обробляє контейнер завдання, чи перебуває в режимі очікування. Таким чином, утримання provisioned concurrency для контейнерів з високою конфігурацією пам'яті, які фактично простоюють, може суттєво збільшувати операційні витрати, що підсилює актуальність задачі оптимального розподілу. Один із підходів до розв’язання цієї проблеми було запропоновано в роботі [151], у якій звернено увагу на те, що оптимальна робота СМО має місце в тому випадку, коли швидкість обробки даних на серверах пропорційна інтенсивності надходження на відповідні сервери, що у випадку однакової кількості серверів та каналів надходження матиме такий вигляд:

$$\frac{\lambda_1}{\mu_1} = \frac{\lambda_2}{\mu_2} = \cdots = \frac{\lambda_c}{\mu_c}. \tag{3.21}$$

В контексті AWS Lambda умова (3.21) означає, що кожна група функцій з певною конфігурацією пам'яті повинна мати provisioned concurrency, пропорційне до інтенсивності надходження завдань відповідного типу. Іншими словами, завантаженість кожної групи контейнерів має бути однаковою, що забезпечує рівномірний розподіл навантаження та запобігає ситуації, коли одні контейнери перевантажені, а інші простоюють.

Розглянемо відповідну задачу знаходження оптимальної кількості серверів, для якої інтенсивності обробки задач будуть пропорційні інтенсивностям надходження задач за різними підпотоками. Для цього розіб'ємо загальний потік $\{1, \dots, k\}$ на l підпотоків $\{C_1, \dots, C_l\}$, причому будемо припускати, що повинна бути виконана умова (3.21). Формально дану задачу можна записати:

$$\min_{l,\{C_1,\dots,C_l\},u} \sum_{i=1}^{l} \left(\frac{\sum_{j \in C_i} w_j \lambda_j}{\mu_i} - u \right)^2 \tag{3.22}$$

за умов, які накладаються виключно за рахунок властивостей серверів. Причому в розгляді відповідних умов можна враховувати наступні:

- Підпотоки повинні враховувати всі вхідні завдання та не можуть перетинатися, тобто:

$$\bigcup_{i=1}^{l} C_i = \{1, \dots, k\}, C_i \cap C_j = \emptyset.$$

- Обмеженість на потужність потоків, що може включати одночасно верхню та нижню межі інтенсивності обробки:

$$\mu_{MIN} \leq \mu_i \leq \mu_{MAX}.$$

- Обмеження на кількість кластерів (підпотоків), які можуть бути визначені при аналізі, а саме:

$$i \in Cl.$$

Для безсерверних обчислень на базі AWS Lambda обмеження набувають конкретного змісту, так μ_{MIN} та μ_{MAX} визначаються мінімальною (128 MB) та максимальною (10240 MB) конфігурацією виділеної пам'яті

Lambda-функцій, оскільки AWS Lambda масштабує CPU пропорційно до виділеної пам'яті, параметр l відповідає кількості груп provisioned concurrency з різними конфігураціями ресурсів, а кластери $\{C_1, \dots, C_l\}$ визначають, які саме типи вхідних завдань обслуговуються кожною групою Lambda-функцій.

Таким чином, розглянута оптимізаційна задача (3.22) із зазначеними вище обмеженнями може значно пришвидшити роботу самої системи з урахуванням оптимального перерозподілу завдань у системі. Крім того, використання такого підходу одразу дозволяє обмежити кількість підпотоків кількістю потоків завдань. З іншого боку, задача (3.22) не є задачею опуклого програмування через наявність μ_i у знаменнику цільової функції, тому підходи опуклого програмування в даному контексті працюватимуть погано. Також задача може бути представлена частково як задача цілочисельного програмування з урахуванням того, що належність до кластера може бути подана у вигляді цілочисельної змінної, значення якої відповідає кластерам, до яких належить цей вхідний потік. Тому для розв’язання цієї проблеми можна скористатися декількома альтернативними підходами:

- Метод грубої сили, якщо кількість можливих варіантів у обмеженнях невелика, з подальшим розв’язанням квадратичної оптимізаційної задачі щодо знаходження параметрів $l, \{C_1, \dots, C_l\}, u$.
- Жадібні алгоритми, які дозволяють також розв’язувати задачі неопуклого програмування із врахуванням різного роду обмежень.
- Генетичні алгоритми можуть бути використані як основний інструмент визначення оптимальних кластерів, тобто виявлення основних підпотоків, а допоміжним або другорядним завданням буде задача підбору інтенсивності обробки завдань $\mu_i,\ i \in Cl$. Ці алгоритми не знаходитимуть глобальний оптимум задачі, проте вони добре пристосовані для знаходження локальних оптимальних значень.

3.3.5 Алгоритм параметричної оцінки конфігурації безсерверної системи

На основі отриманих теоретичних результатів (Лема 3.2.1, Теорема 3.2.3, формули (3.13), (3.17)) пропонується алгоритм параметричної оцінки та оптимізації конфігурації безсерверної системи. Алгоритм використовує метрики AWS CloudWatch як джерело вхідних даних та результати напівмарковської моделі для прийняття рішень щодо кількості функцій, що можуть миттєво обробляти запити (provisioned concurrency). Відмінність від існуючих підходів на основі машинного навчання [160, 167] полягає у використанні аналітичних оцінок, що дозволяє отримати явні формули для оптимальних параметрів.

Крок 1. Ідентифікація джерел та збір даних.

На основі метрик AWS CloudWatch (Invocations, ConcurrentExecutions, Duration, Throttles) оцінити параметри вхідного процесу:

- інтенсивності $\lambda_i(t)$, $i = 1, \dots, k$, для кожного тригера (API Gateway, S3, SQS, EventBridge тощо) на основі метрики Invocations з розбивкою по джерелу. Для тригерів типу SQS додатково використовуються метрики черги такі як *NumberOfMessagesSent* для оцінки $\lambda_i(t)$ та *NumberOfMessagesReceived* для оцінки ефективної інтенсивності обробки. Метрика *ApproximateNumberOfMessagesVisible* [168] відповідає поточній довжині черги $L(t)$ і може використовуватися для валідації прогнозованих значень $\rho_0(\Delta, t)$;
- перевірити обмеженість інтенсивностей: $\lambda_i(t) \leq \lambda_{\text{MAX}}$ для всіх i, t.

Крок 2. Оцінка параметрів обробки.

Визначити інтенсивності обробки μ_i, $i = 1, \dots, n$, для кожної конфігурації Lambda-функції:

$$\mu_i = \frac{1}{E[\text{Duration}_i]},$$

де $E[\text{Duration}_i]$ – середній час виконання функції з i-ою конфігурацією пам'яті c_i^{mem}, що визначається з метрики Duration. Визначити множину активних серверів $A(t)$ та відповідну сумарну інтенсивність обробки

$$\mu(t) = \sum_{i \in A(t)} \mu_i.$$

Крок 3. Перевірка умови стійкості системи (існування невиродженого стаціонарного розподілу).

Обчислити усереднені інтенсивності за формулами:

$$\Lambda(T) = \frac{1}{T} \int_0^T \sum_{i=1}^{k} w_i\,(t) \lambda_i(t)\, dt,$$

$$M(T) = \frac{1}{T} \int_0^T \sum_{i \in A(t)} \mu_i\ dt$$

та перевірити виконання умови $\Lambda(T)/M(T) < 1$. Якщо умова порушена – система потребує збільшення кількості серверів або зменшення вхідного навантаження перед подальшою оптимізацією.

Крок 4. Обчислення рівноважної функції $\rho_0(\Delta, t)$***.***

Для заданого кроку дискретизації Δ (що відповідає інтервалу моніторингу CloudWatch: 1с для детального моніторингу або 60с для базового моніторингу) знайти $\rho_0(\Delta, t)$ як розв'язок рівняння

$$\phi(\rho_0, \Delta, t) = E(L(t+\Delta) - L(t) \mid L(t) = \rho_0) = 0$$

для кожного $t \in [0, T]$. Розв'язок знаходиться чисельно методом бісекції, оскільки $\phi(k, \Delta, t)$ є строго спадною функцією по k (Лема 3.2).

Крок 5. Обчислення параметрів дифузійної апроксимації.

Обчислити дифузійний коефіцієнт $B(\rho_0(\Delta, t), t)$ та дисперсію

$$\sigma^2(t) = \lim_{\varepsilon \to 0} \frac{1}{2\varepsilon^2} \int_0^{\varepsilon^2 t} B\,(\rho_0(\Delta, s), s)\, ds,$$

де параметр $\varepsilon > 0$ визначає масштаб дифузійної апроксимації. На практиці ε обирається з умови $\varepsilon^2 T \sim T_{\text{obs}}$, де T_{obs} – тривалість періоду спостереження.

Крок 6. Оптимізація конфігурації.

Розв’язати задачу щодо визначення кількісті зарезервованих контейнерів з обмеженням на показник відхилення запитів (3.20):

$$\sum_{i=1}^{n} c_i \cdot m_i \to \min$$

за умови $\theta(T) \leq \theta_{\text{SLA}}$ та $\Lambda(T)/M(T) < 1$.

Результатом є оптимальна конфігурація: кількість екземплярів, що можуть миттєво обробляти запити (provisioned concurrency) m_i^* для кожної групи Lambda-функцій.

Крок 7. Валідація.

Перевірити отриману конфігурацію $\{m_i^*\}$ шляхом:

- порівняння прогнозованих значень $\rho_0(\Delta, t)$ та $\sigma(t)$ з реальними метриками ConcurrentExecutions;
- порівняння показника відхилення запитів $\theta(T)$ з реальною метрикою Throttles;
- за наявності суттєвих розбіжностей – повернення до Кроку 1 з уточненими параметрами.

Крок 8. Періодична переоцінка параметрів.

Визначити інтервал переоцінки $T_{\text{re}} > 0$ та повторювати Кроки 1–7 з періодичністю T_{re}. Інтервал T_{re} обирається з урахуванням двох факторів:

- *нижня межа*: $T_{\text{re}} \geq T_{\min}$, де $T_{\min}$ визначається мінімальним обсягом даних, необхідним для статистично значущої оцінки параметрів $\lambda_i(t)$ та μ_i;
- *верхня межа*: $T_{\text{re}} \leq T_{\max}$, де $T_{\max}$ визначається характерним часом зміни параметрів системи (добовий цикл навантаження, період між деплоями коду).

При переоцінці на інтервалі $[T_n, T_n + T_{\text{re}}]$, де $T_n = n \cdot T_{\text{re}}$, параметри оновлюються:

$$\hat{\lambda}_i^{(n)} = \frac{1}{T_{\text{re}}} \int_{T_n}^{T_n + T_{\text{re}}} \lambda_i(t)\, dt, \quad \hat{\mu}_i^{(n)} = \frac{1}{T_{\text{re}}} \int_{T_n}^{T_n + T_{\text{re}}} \mu_i(t)\, dt,$$

де $\lambda_i(t)$ та $\mu_i(t)$ оцінюються на основі метрик CloudWatch (Invocations, Duration) та SQS (NumberOfMessagesSent, NumberOfMessagesReceived) за відповідний інтервал. Конфігурація $\{m_i^*\}$ оновлюється лише за умови

$$\left|\frac{\hat{\lambda}_i^{(n)} - \hat{\lambda}_i^{(n-1)}}{\hat{\lambda}_i^{(n-1)}}\right| > \delta_\lambda \quad \text{або} \quad \left|\frac{\hat{\mu}_i^{(n)} - \hat{\mu}_i^{(n-1)}}{\hat{\mu}_i^{(n-1)}}\right| > \delta_\mu,$$

де $\delta_\lambda, \delta_\mu > 0$ – пороги значущості зміни (типово $\delta_\lambda = \delta_\mu = 0{,}1$), що дозволяє уникнути надмірно частих переконфігурацій системи.

Таким чином, послідовність оцінок $\left\{\hat{\lambda}_i^{(n)}, \hat{\mu}_i^{(n)}, m_i^{*(n)}\right\}_{n\geq 0}$ формує адаптивний процес управління конфігурацією безсерверної системи. Перевагою запропонованого підходу порівняно з існуючими методами на основі машинного навчання є відсутність необхідності перенавчання моделі (оновлення параметрів здійснюється безпосередньо на основі аналітичних формул).

Зауважимо, що аналогічний ітеративний підхід до оптимізації конфігурації безсерверних систем на основі прогнозування навантаження запропоновано у роботі [164], де використовується динамічний вибір моделі для прогнозування навантаження. Відмінність запропонованого алгоритму полягає у використанні аналітичних оцінок на основі теорії масового обслуговування, що дає змогу отримати явні формули для оптимальних параметрів та уникнути необхідності навчання ML-моделей і використання нейронних мереж.

Висновки до розділу 3

У третьому розділі побудовано математичну модель безсерверної інформаційної системи на прикладі AWS Lambda як неоднорідної СМО зі змішаними режимами роботи та розроблено методи оцінювання її ключових параметрів.

Проведено огляд та систематизацію існуючих підходів до оцінювання параметрів СМО складної структури, зокрема результатів щодо центральної

граничної теореми для СМО з великою кількістю серверів, дифузійних апроксимацій, нормальних наближень для замкнених та відкритих СМО, а також задач оптимізації кількості серверів при різних критеріях якості обслуговування.

Встановлено необхідну та достатню умову обмеженості черги (Лема 3.2.1): середня інтенсивність надходження має бути строго меншою за середню інтенсивність обробки. У контексті AWS Lambda ця умова визначає мінімальну пропускну здатність системи, що гарантує стабільну роботу без необмеженого зростання затримок та помилок щодо відхилення завдань.

Досліджено властивості умовного математичного сподівання приросту довжини черги (Лема 3.2.2) та встановлено існування точки рівноваги, що визначає типовий рівень навантаження системи та є орієнтиром для налаштування механізмів автоматичного масштабування.

Доведено граничні еволюції для процесу довжини черги у схемі усереднення та схемі дифузійної апроксимації (Теорема 3.2.3) з використанням апарату напівмарковських випадкових еволюцій. Схема усереднення визначає детерміновану траєкторію середнього навантаження Lambda-функції, а схема дифузійної апроксимації описує випадкові відхилення від цієї траєкторії через стохастичне диференціальне рівняння Ito.

Розроблено метод оцінки параметрів суміші вхідного процесу $(w_i(t), \lambda_i(t))$ на основі EM-алгоритму з використанням метрик AWS CloudWatch та SQS. Досліджено два протоколи розподілу завдань між серверами – випадковий та оптимальний (вибір найпотужнішого вільного сервера). Для *M/M/2* системи аналітично доведено, що оптимальний протокол забезпечує меншу середню довжину черги ($L_q^{opt} < L_q$) та менший час очікування ($W_q^{opt} < W_q$). Сформульовано задачу оптимального розподілу вхідних потоків між групами серверів з різною потужністю за критерієм рівномірного завантаження.

Запропоновано алгоритм параметричного оцінювання безсерверної системи, що складається з восьми кроків: збір даних з AWS CloudWatch, оцінка параметрів обробки, перевірка умови стабільності, обчислення рівноважної функції, оцінка параметрів дифузійного наближення, оптимізація конфігурації та валідація. Відмінність запропонованого підходу від існуючих методів на основі машинного навчання полягає у використанні аналітичних оцінок на основі теорії масового обслуговування, що дозволяє отримати явні формули для оптимальних параметрів.

Основні результати розділу опубліковані у працях [103, 129, 153].

РОЗДІЛ 4
ДОСЛІДЖЕННЯ СТВОРЕНОЇ СИСТЕМИ, ПРАКТИЧНА РЕАЛІЗАЦІЯ ТА РЕЗУЛЬТАТИ ОПТИМІЗАЦІЇ

У розділі наведено результати тестування запропонованого у дисертаційній роботі фреймворку для оптимізації розподілених безсерверних обчислень у хмарних середовищах. Метою тестування є перевірка ефективності розробленої архітектури та алгоритмів автоматичного масштабування обчислювальних ресурсів при обробці великого потоку завдань. Тестування виконано на основі імітаційного моделювання та двох серій експериментів.

У першій серії експериментів досліджено прогнозне автоматичне масштабування обчислювальних ресурсів на основі моделі DeepAR, яка використовує метрики SQS для прогнозування навантаження за допомогою рекурентних нейронних мереж. Друга серія експериментів стосувалася прогнозного автомасштабування на основі аналітичної моделі з напівмарковськими процесами, що безпосередньо базується на результатах Розділу 3. Також проведено імітаційне моделювання методом Монте-Карло для верифікації теоретичних результатів щодо відмінності моделі суміші потоків від MMPP-наближення та переваги оптимального протоколу вибору серверу.

Порівняння результатів трьох експериментів дозволяє оцінити переваги та обмеження кожного підходу та обґрунтувати вибір моделі, що здатна покращувати реальні показники продуктивності інформаційної системи.

4.1 Прогнозне автоматичне масштабування на основі DeepAR

Метою експерименту є дослідження ефективності автоматичного масштабування обчислювальних ресурсів у безсерверних розподілених системах з використанням черг повідомлень. Підхід передбачає перехід від

реактивного до проактивного масштабування, що дозволяє зменшити кількість «холодних стартів» та помилок перевантаження [111, 169].

Даний експеримент безпосередньо пов'язаний з теоретичною моделлю $H_k(t)/M/\infty$, побудованою у Розділі 3. Метрики AWS SQS, що використовуються для прогнозування, мають пряму відповідність параметрам моделі: ApproximateNumberOfMessagesVisible відповідає процесу довжини черги $L(t)$, NumberOfMessagesSent відповідає інтенсивності надходження заявок $\lambda(t)$ (формула (3.4)), а NumberOfMessagesReceived – інтенсивності обробки $\mu(t)$. Відношення цих метрик на інтервалі $[0, T]$ дає емпіричну оцінку $\frac{\Lambda(T)}{M(T)}$, що використовується для перевірки умови стабільності (Лема 3.2.1). Модель $H_k(t)/M/\infty$ використовується для аналізу граничної поведінки системи та отримання аналітичних оцінок. У практичній реалізації кількість серверів обмежена квотою Lambda (параметр max_c в Algorithm 2), що відповідає моделі з кінцевою кількістю каналів обслуговування. Перехід від теоретичної моделі M/∞ до кінцевої системи здійснюється через формулу throttling rate (3.20), яка явно враховує обмеженість ресурсів.

Для цього розроблено і розгорнуто прототип розподіленого застосунку, призначений для пакетної обробки даних в AWS, який використовується для прогнозування навантаження та автоматичного масштабування ресурсів. Головною особливістю застосунку є використання безсерверних технологій. Основним завданням розробленого прототипу є забезпечення розподіленої обробки великих обсягів даних з мінімальною кількістю помилок. У таких системах черга, зазвичай AWS SQS, є основним компонентом, який отримує події та розподіляє дані на встановлену максимальну кількість обробників, як AWS Lambda [71]. Залежно від поточного навантаження запускається лише та кількість обробників, яка потрібна в певний момент часу в межах квоти. Вхідні дані та результати обробки зберігаються на S3.

Для моніторингу стану черги використовується AWS CloudWatch, який відстежує AWS SQS, а саме метрики використання ресурсів. З точки зору проблеми автоматичного масштабування, цікавими є дві метрики:

• ApproximateNumberOfMessagesVisible – довжина черги.

• NumberOfMessagesSent – кількість надісланих повідомлень.

Для автоматичного масштабування розробленого застосунку використовувався AWS SageMaker [109]. Цей безсерверний сервіс дозволяє швидко створювати та розгортати моделі машинного навчання з мінімальним втручанням розробника. Також розгорнуто декілька компонентів для взаємодії з AWS SQS. Перш за все, це AWS CloudWatch, який збирає необхідні метрики з AWS SQS. AWS EventBridge [110] запускає AWS Lambda [108] через відповідні проміжки часу згідно з розкладом, а AWS Lambda отримує прогнозовані значення з навченої моделі та коригує кількість та основні параметри обробників даних, таким чином забезпечуючи горизонтальне масштабування.

Взаємодія між зазначеними компонентами та розподіленим застосунком відбувається наступним чином:

1. AWS CloudWatch збирає необхідні метрики з AWS SQS та зберігає їх у S3.
2. Модель машинного навчання створюється та розгортається в AWS SageMaker на основі збережених метрик.
3. AWS EventBridge запускає AWS Lambda за заздалегідь визначеним розкладом.
4. AWS Lambda отримує доступ до розгорнутої моделі та отримує прогнозовані показники навантаження.
5. На основі отриманих показників AWS Lambda відповідно змінює налаштування обробників даних.

Модель побудована за допомогою DeepAR та алгоритму навчання для прогнозування часових рядів з використанням рекурентних нейронних мереж (RNN) [169, 170]. Цей алгоритм може забезпечити кращу точність, ніж

класичні методи прогнозування, такі як авторегресивні інтегровані ковзні середні (ARIMA) або експоненціальне згладжування (ES) [171, 172]. Загалом DeepAR підвищує точність прогнозування, досліджуючи закономірності з кількох пов'язаних часових рядів, на відміну від традиційних методів, які спираються на один часовий ряд. DeepAR створює точкові та ймовірнісні прогнози, що особливо важливо для планування потужностей [173]. Точність розподілу прогнозів оцінюється за допомогою зважених квантильних втрат. Квантильна втрата для квантиля q∈(0,1) розраховується наступним чином.

$$QL(q, y, \hat{y}) = \begin{cases} q \cdot (y - \hat{y}), & \text{якщо } y > \hat{y} \\ (1 - q) \cdot (\hat{y} - y), & \text{якщо } y \leq \hat{y} \end{cases} \quad (4.1)$$

де:

q – обраний квантиль (наприклад, 0.1, 0.5, or 0.9),

y – фактичне значення,

$\hat{y}$ – прогноз для відповідного квантиля.

Для набору квантилів $Q = \{q_1,\ q_2, \dots,\ q_k\}$ загальна зважена квантильна втрата розраховується як (4.2):

$$QL_{weighted} = \frac{1}{n} \sum_{i=1}^{n} \sum_{q \in Q} \omega_q \cdot QL\left(q, y_i, \hat{y}_i^{(q)}\right) \quad (4.2)$$

де [170]:

n – кількість прогнозованих очок;

ω_q – вага для квантиля q;

$\hat{y}_i^{(q)}$ – прогноз для квантиля q для точки i.

Також використовувалися середня абсолютна похибка (4.3) та середньоквадратична похибка (4.4):

$$MAE = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i|, \quad (4.3)$$

$$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2}. \quad (4.4)$$

Побудова моделі DeepAR складається з таких кроків:

1. Збір та форматування метрик AWS SQS.

2. Визначення та навчання гіперпараметрів.

3. Навчання моделі.

4. Тестування моделі.

5. Оцінка результатів.

6. Розгортання моделі для подальшого використання.

Отримані результати прогнозування використовуються для автоматичного масштабування обробників даних відповідно до наведеного нижче алгоритму [111] з використанням порогових значень.

Algorithm 2. Коригування зарезервованої (provisioned) конкурентності в AWS Lambda

```
1: function AutoScaleLambda(f_name, t_up, t_down, min_c,
max_c, inc, target)
2:    loop every N minutes
3:       visible ← GetPrediction(target)
4:       current_c ← GetProvisionedConcurrency(f_name)
5:
6:       if visible > t_up then
7:          new_c ← current_c + inc
8:          if new_c > max_c then
9:             new_c ← max_c
10:         end if
11:         SetProvisionedConcurrency(f_name, new_c)
12:      else if visible < t_down then
13:         new_c ← current_c - inc
14:         if new_c < min_c then
15:            new_c ← min_c
16:         end if
17:         SetProvisionedConcurrency(f_name, new_c)
18:      end if
19:      Sleep(N minutes)
20:   end loop
21: end function
```

де:

f_name – ім’я Lambda функції,

t_up – поріг масштабування для ApproximateNumberOfMessagesVisible,

t_down – поріг зменшення масштабу для ApproximateNumberOfMessagesVisible,

min_c – мінімальна provisioned concurrency,

max_c – максимальна provisioned concurrency,

inc – крок збільшення масштабування,

target – значення історичних часових рядів.

Побудований таким чином фреймворк автоматично масштабуватиме обробники даних за допомогою черг [169]. Зауважимо, що порогові значення t_up та t_down в Algorithm 2 визначаються емпірично, що є основним обмеженням цього підходу. Як буде показано у підпункті 4.2, ці пороги можуть бути замінені на аналітично обґрунтовані значення на основі результатів Розділу 3.

Для моделювання робочого навантаження використано Locust. Цей інструмент ефективний для навантажувального тестування розподілених систем завдяки своїй масштабованості, простоті написання сценаріїв, а також можливостям моніторингу та аналізу [173].

Під час тестування застосунку черга отримала 1082178 подій протягом 90 хвилин. Водночас збиралися метрика приблизної кількості видимих повідомлень та метрика кількості надісланих повідомлень для AWS SQS [129]. На рисунках 4.1 та 4.2 наведено розподіл зібраних даних.

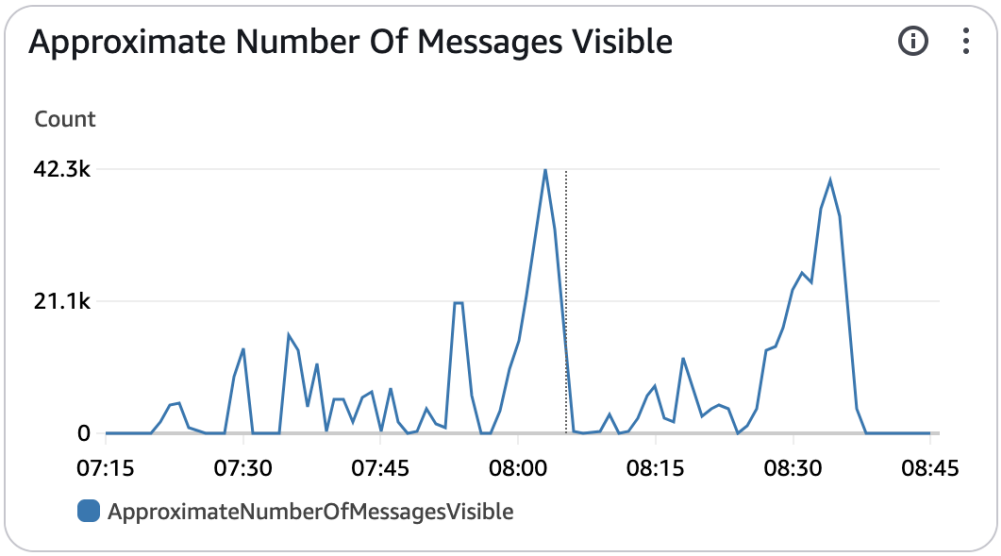

Рис. 4.1 – Приблизна кількість видимих повідомлень

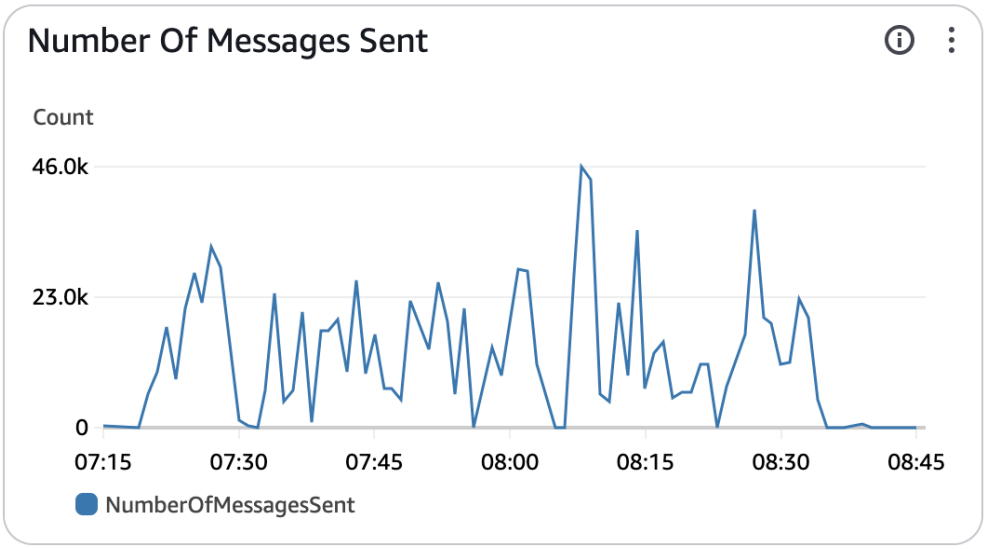

Рис. 4.2 – Кількість відправлених повідомлень

Дані демонструють характерну сплескову поведінку, що є типовою для безсерверних систем з event-driven архітектурою.

Зібрані метрики використовувалися для навчання моделі DeepAR. Варто зазначити, що DeepAR використовує рекурентну нейронну мережу LSTM (RNN). Для навчання обрано похвилинну гранулярність часових рядів. Модель потребувала щонайменше 30 кроків історичних даних для прогнозування з початковою тривалістю прогнозу 50 хвилин. Для навчання моделі використано конфігурацію AWS SageMaker з параметрами за замовчуванням, за винятком кількості шарів RNN, яка варіювалася від 1 до 8

для визначення оптимальної архітектури. Вибір решти гіперпараметрів за замовчуванням обумовлений тим, що метою даного експерименту є не побудова оптимальної ML-моделі прогнозування, а порівняння ML-підходу з аналітичним підходом на основі теорії масового обслуговування, запропонованим у Розділі 3. Детальне налаштування гіперпараметрів DeepAR не впливає на основний висновок дисертації – перевагу аналітичного підходу, який не потребує навчання моделі.

Результати навчання представлені на рисунках 4.3 та 4.4.

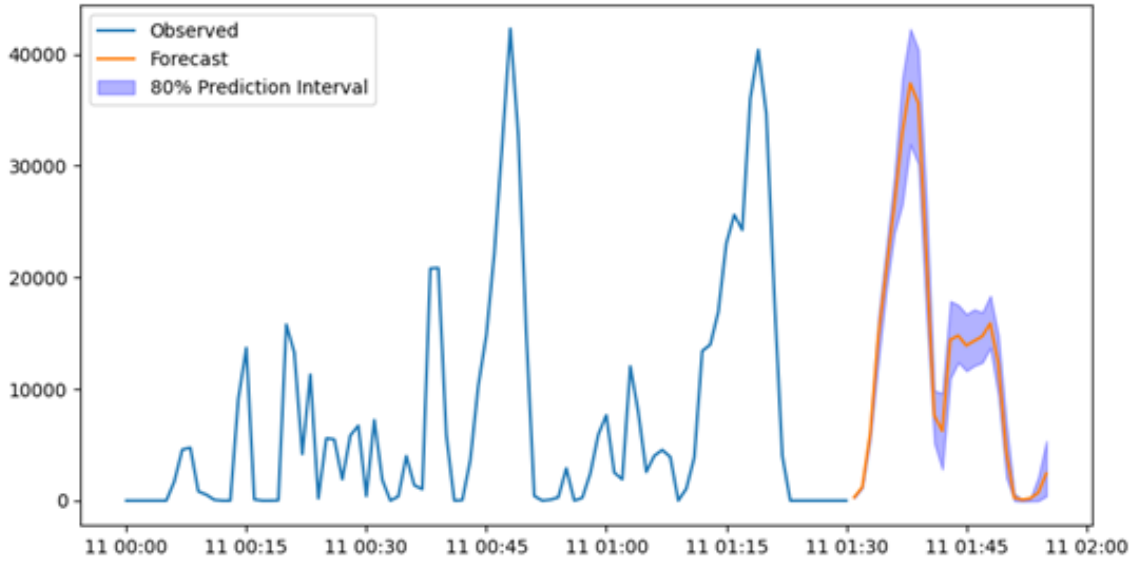

Рис. 4.3 – Прогнозовані значення приблизної кількості видимих повідомлень

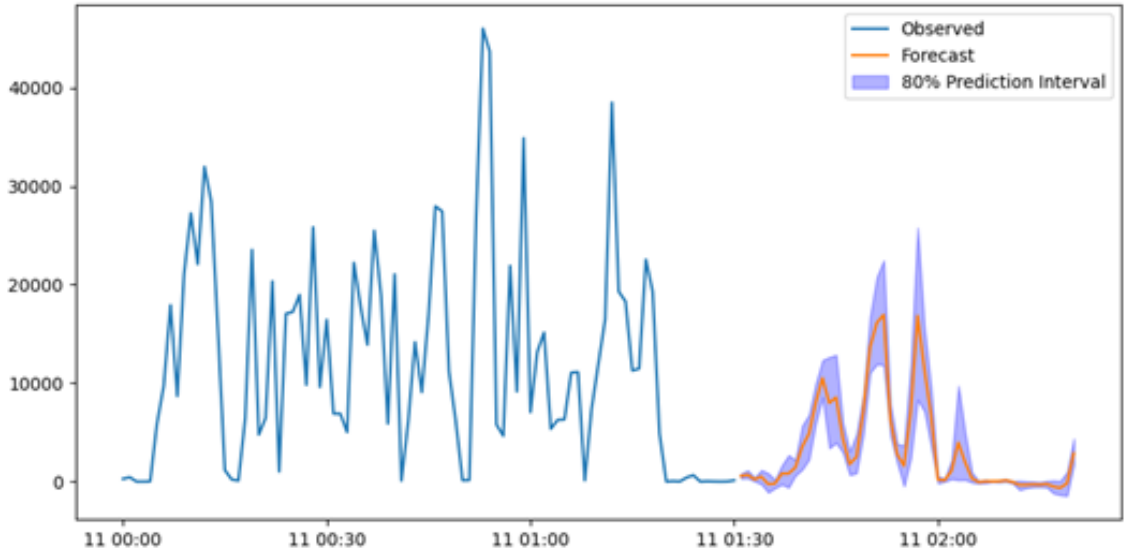

Рис. 4.4 – Прогнозовані значення кількості надісланих повідомлень

Для оцінювання отриманих моделей було використано метрики середньої абсолютної похибки та кореня середньоквадратичної похибки для різної кількості шарів. На рисунках 4.5 та 4.6 наведено порівняння отриманих значень похибок.

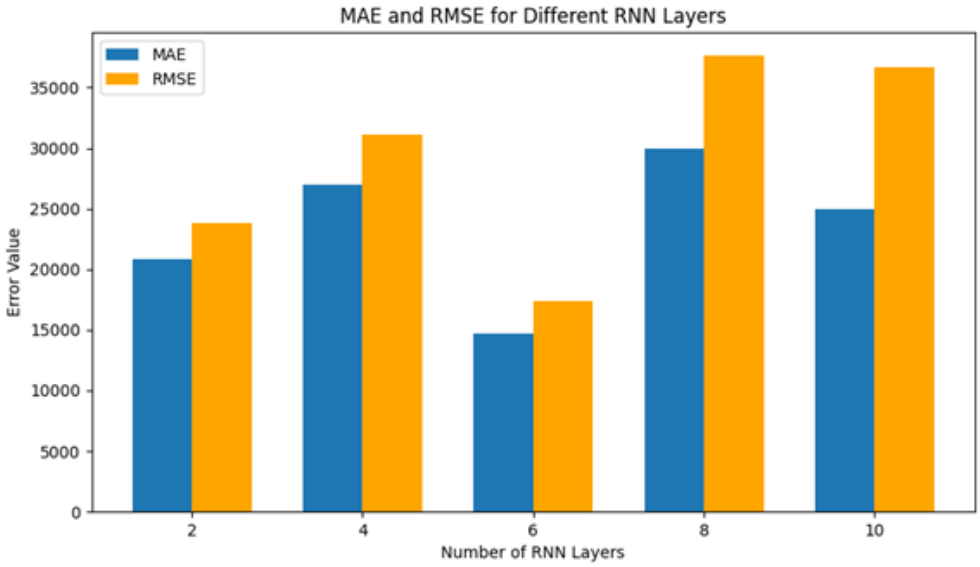

Рис. 4.5 – MAE та RMSE для різної кількості шарів RNN при прогнозуванні приблизної кількості видимих повідомлень

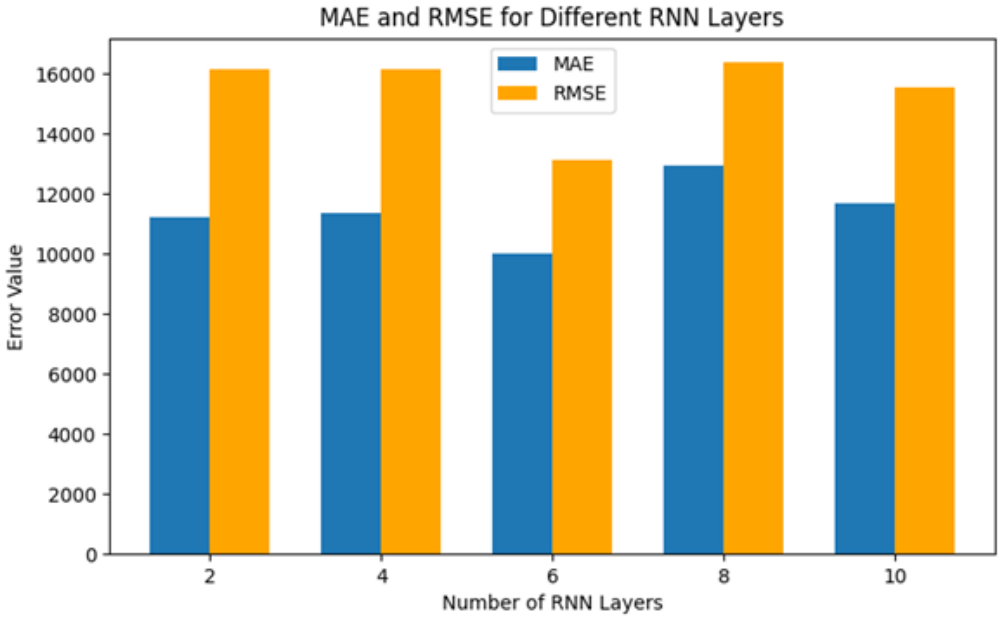

Рис. 4.6 – Значення MAE та RMSE для різної кількості шарів RNN при прогнозуванні кількості надісланих повідомлень

У результаті варіювання кількості шарів встановлено, що моделі демонструють найкращу продуктивність при шести шарах і можуть

використовуватися для прогнозування навантаження розподіленого застосунку. Запропонований підхід забезпечує перехід від реактивного до проактивного масштабування, що дозволяє зменшити кількість «холодних стартів» та помилок перевантаження. Модель DeepAR демонструє середню абсолютну похибку (MAE) 10019.21 та корінь із середньоквадратичної похибки (RMSE) 13140.91. При середньому значенні метрики ApproximateNumberOfMessagesVisible 587 254 за період тестування відносна похибка прогнозу становить приблизно 1,7%, а нормалізований RMSE – 2,2%, що свідчить про достатньо високу якість прогнозування для задач проактивного масштабування. Застосування прогнозування для автоматичного масштабування обробників даних дало змогу зменшити кількість «холодних стартів» приблизно на 27%, а кількість необроблених запитів – приблизно на 14% [111, 169].

Основним обмеженням запропонованого підходу є недостатня кількість наявних історичних даних, необхідних моделі DeepAR для виявлення закономірностей. У випадках, коли часовий ряд містить дуже мало попередніх спостережень (наприклад, для новостворених сервісів), модель може неточно відображати тенденції, що знижує достовірність прогнозування. Зазначені обмеження підходу на основі DeepAR є причиною дослідження альтернативного підходу, який базується на запропонованій нами аналітичній моделі, описаній у Розділі 3. На відміну від нейронних мереж, аналітичний підхід не потребує великого обсягу історичних даних для навчання, оскільки параметри моделі $\left(w_i(t), \mu_i(t), \lambda_i(t)\right),\ t \geq 0$ оцінюються безпосередньо з поточних метрик CloudWatch та SQS. Крім того, аналітичний підхід дає явні формули для оптимальних параметрів provisioned concurrency та забезпечує аналітичні гарантії SLA через формулу throttling rate (3.20), що є принциповою перевагою перед евристичними порогами Algorithm 2.

4.2 Прогнозне автоматичне масштабування на основі аналітичної моделі

Метою другого експерименту є практична перевірка ефективності прогнозного автоматичного масштабування з використанням марковських моделей, що безпосередньо базуються на теоретичних результатах Розділу 3. Для тестування створено безсерверний додаток на основі розробленого фреймворку [103]. Додаток розгорнуто на AWS у двох версіях: з прогнозним автоматичним масштабуванням та без нього (класичне реактивне масштабування). Архітектура фреймворку та модуль прогнозного автомасштабування детально описані у підпункті 2.4 (рис. 2.5–2.10). У даному підпункті описано результати тестування фреймворку з модулем, що використовує напівмарковські моделі (Algorithm 1 з підпункту 2.4), які безпосередньо базуються на теоретичних результатах Розділу 3.

Навантажувальне тестування розподіленої системи проводилося з використанням фреймворку Locust. Архітектура додатка повністю використовує можливості фреймворку, де основним компонентом є черга повідомлень, яка забезпечує слабку зв’язність інших компонент додатка. Це, своєю чергою, дозволяє легко масштабувати потрібні частини системи, швидше відновлюватися від збоїв та більш ефективно використовувати наявні ресурси.

В якості вхідних даних використовувалися файли у форматі csv, які містили 1025132 записи. На основі кожного запису генерувався односторінковий PDF документ. Записи розділялися на групи розміром по 50 елементів в групі та надсилалися до черги із затримкою в 1 секунду. Для обробників даних обрано мінімальну конфігурацію пам'яті AWS Lambda (128 MB) з середовищем виконання Node.js v22. Даний вибір обґрунтовано на основі аналізу залежності вартості виконання від конфігурації ресурсів з Розділу 3.

Вартість виконання одного запиту Lambda визначається формулою з підпункту 3.3.2

$$c_i^{\text{req}} = p_{\text{GB-s}} \cdot c_i^{\text{mem}} \cdot \mu_i^{-1} + p_{\text{req}}, \quad (4.5)$$

де $p_{\text{GB-s}}$ = 0,0000166667 USD/GB-c – вартість обчислень (USD за GB-секунду), c_i^{mem} – обсяг виділеної пам'яті i-го контейнера (GB), μ_i^{-1} – середній час обробки одного запиту (c), p_{req} = 0,0000002 USD – вартість одного запиту. Lambda масштабує CPU пропорційно виділеній пам'яті, тому збільшення пам'яті прискорює обробку і, як наслідок, виникає питання чи може таке прискорення компенсувати збільшення вартості за одиницю часу.

Для відповіді на це питання проведено моделювання залежності загальної вартості обробки 1 025 132 записів від конфігурації пам'яті. Так, залежність часу виконання від конфігурації ресурсів визначається на основі моделі [174], де інтенсивність обробки μ_i пов'язана з виділеними ресурсами степеневою залежністю з параметром α. Оскільки $\mu_i = \frac{1}{T_i}$, час виконання при конфігурації пам'яті m визначається як $T(m) = T_{base}(\frac{m_{base}}{m})^{\alpha}$. Параметр α визначається емпірично та залежить від типу навантаження. Для генерації PDF-документів, що є змішаним I/O та CPU навантаженням, $\alpha \approx 0.7$.

Результати моделювання (таблиця 4.1) демонструють, що при $\alpha \leq 1$ мінімальна конфігурація є оптимальною за вартістю. Збільшення пам'яті прискорює обробку, але не компенсує зростання вартості за GB/секунду.

Таблиця 4.1.

Вартість обробки 1 025 132 записівпри різних конфігураціях Lambda

Конфігурація	Час виконання (с)	μ_i (задач/с)	Вартість/ запит ($)	Загальна вартість ($)	Δ від мін.*
128 MB	2,900	0,345	0,0000062	0,13	–
256 MB	1,785	0,560	0,0000076	0,16	+22%
512 MB	1,099	0,910	0,0000094	0,19	+50%
1024 MB	0,676	1,478	0,0000115	0,24	+84%
1769 MB (1 vCPU)	0,461	2,168	0,0000138	0,28	+121%
3072 MB	0,314	3,190	0,0000159	0,33	+154%

* – відсоткова різниця загальної вартості обробки кожної конфігурації відносно мінімальної конфігурації (128 MB)

Разом з тим, моделювання показує, що при $\alpha > 1$ (масштабування, можливе для задач з інтенсивним використанням кешу L2/L3 або паралелізму на рівні інструкцій) потужніша конфігурація стає дешевшою: при $\alpha = 1{,}1$ максимальна конфігурація (3072 MB) забезпечує економію 26,4% порівняно з мінімальною. Таким чином, вибір мінімальної конфігурації для задачі генерації PDF є обґрунтованим. Для задач з іншими характеристиками рекомендується емпірично визначити α шляхом вимірювання часу виконання при двох конфігураціях та використати формулу (4.5) для визначення оптимальної конфігурації.

Під час тестування збиралися та аналізувалися кілька найбільш критичних показників, таких як: час затримки даних в обробці, кількість повідомлень, оброблених за одиницю часу, кількість холодних стартів.

На рис. 4.7 показано зміну часу обробки даних у розробленому додатку протягом 5 хвилин. Варто зазначити, що використання прогнозного автоматичного масштабування обчислювальних ресурсів пришвидшило обробку даних в середньому на 25,8%.

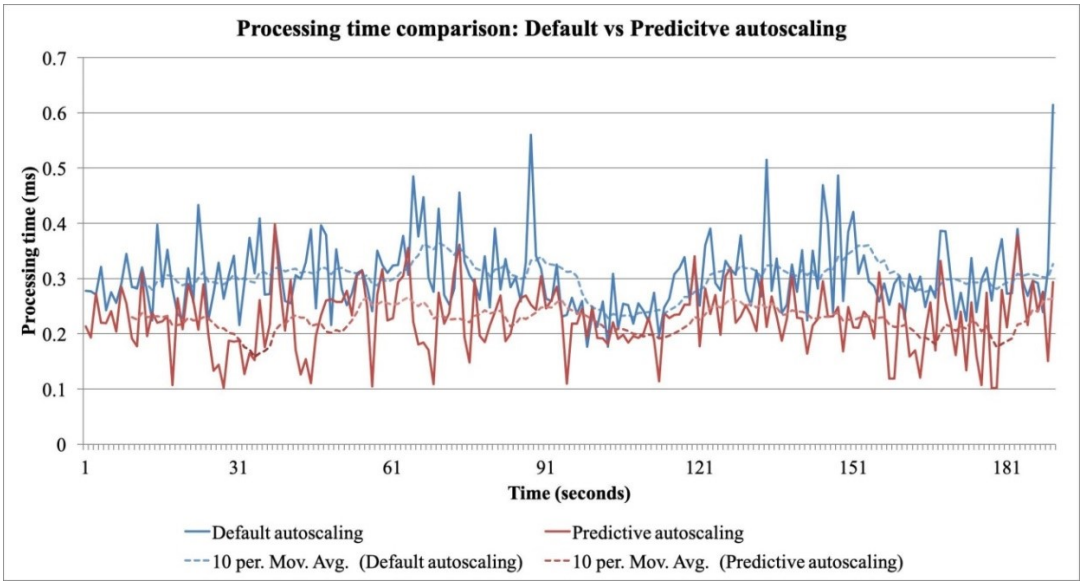

Рис. 4.7 – Час оброблення даних із використанням прогнозного автоматичного масштабування та масштабування за замовчуванням

Можливість динамічно коригувати конфігураційні параметри SQS і Processing Lambda призвела до збільшення пропускної здатності системи в

середньому на 21,3%.

Порівняльний графік на рис. 4.8 демонструє зміну поведінки додатку з використанням прогнозного автомасштабування. Як видно з рис. 4.8 реактивне масштабування не завжди програє прогнозному, тобто існують інші чинники, вплив яких не дозволяє гарантовано отримати необхідний результат.

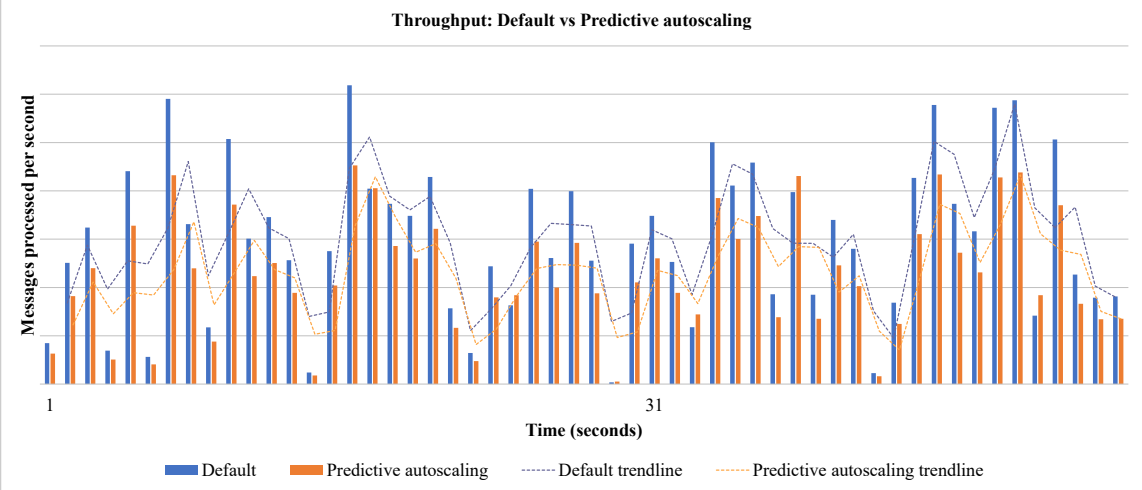

Рис. 4.8 – Пропускна здатність з використанням прогнозного автоматичного масштабування та масштабування за замовчуванням

Одним із таких чинників є проблема холодного старту. Незважаючи на те, що прогнозне автомасштабування дозволило зменшити частку холодних стартів приблизно до 3% (рис. 4.9), повністю позбутися негативного ефекту від цього явища не вдалося.

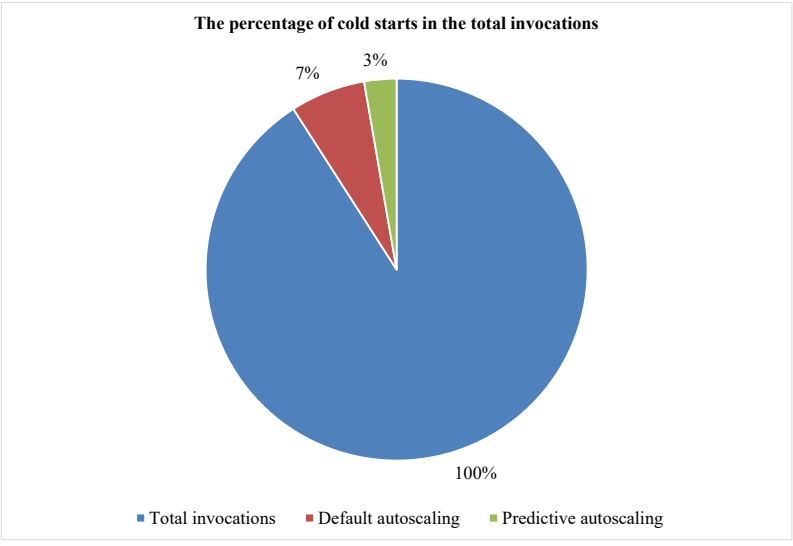

Рис. 4.9 – Відсоток холодних стартів у загальній кількості викликів

Загалом прогнозне автоматичне масштабування стабілізувало поведінку безсерверного додатка з точки зору використання ресурсів, особливо у випадках різкої зміни навантаження, завдяки, в першу чергу, можливості своєчасно приймати рішення щодо масштабування.

У ході дослідження основна увага приділялася ізоляції окремих складових фреймворку для покращення масштабованості системи та спрощення її супроводу. Всі компоненти фреймворку є безсерверними і потребують мінімального втручання розробників.

Реалізація прогнозного автоматичного масштабування у вигляді окремого модуля у складі фреймворку дозволила адаптуватися до пікових навантажень та змінювати налаштування черг і обробників даних на основі прогнозу та визначених наперед порогових значень.

Використання фреймворку для розробки прототипів безсерверних застосунків значно прискорило обробку даних – приблизно на 25,8% (рис. 4.7). Це насамперед пов'язано з можливістю своєчасно виділяти або вивільняти обчислювальні ресурси в умовах змінного навантаження. На відміну від традиційних методів [53, 54.], де масштабування відбувається з певною затримкою, фреймворк покращує цей процес, що є критично важливим для безсерверних рішень.

Як видно з рис. 4.8, пропускна здатність системи зросла приблизно на 21,3%. На відміну від додатків, що конфігуруються статично [100], зміни в конфігурації параметрів черги й обробників даних безпосередньо під час виконання дозволяють вчасно реагувати на піки навантаження та зменшити час між викликами функцій. Ще одним аспектом, який має позитивний вплив на пропускну здатність системи, є використання багаторівневої архітектури додатка та черг повідомлень. Такий підхід дозволяє реалізувати слабку зв’язність між окремими елементами системи та незалежно їх масштабувати в разі потреби.

Для зменшення кількості холодних стартів функцій Lambda в роботах [14, 15, 44, 175] досліджуються різноманітні техніки, такі як періодичне

прогрівання обробників даних за певним розкладом, застосування оптимізацій для зменшення часу ініціалізації, використання кешування, виділення зарезервованих екземплярів обробників, використання машинного навчання та ін. На відміну від робіт [15, 175], які пропонують гібридний механізм прогрівання функцій-обробників на основі шаблонів в історичних даних і використання класифікаторів для визначення та запуску критичних функцій, запропонований підхід (прогнозне автомасштабування) зменшив частку холодних стартів до 3% (рис. 4.9). Отриманий результат є порівнянним із досягненнями гібридних методів. Запропонований фреймворк є доповненням та розширенням вищезазначених досліджень у контексті використання безсерверних обчислень і, на відміну від них, дає змогу зменшити обчислювальну складність, а також не вимагає наявності історичних даних [103].

Результати дослідження дозволяють стверджувати, що головна мета, а саме створення фреймворку для розподіленої обробки даних з використанням прогнозного автоматичного масштабування обчислювальних ресурсів, досягнута. Проблема превентивного масштабування ресурсів для покращення найбільш критичних показників системи вирішена частково. Особливістю розробленого фреймворку є можливість бути частиною будь-якого безсерверного додатка для розподіленої обробки даних з використанням черг.

Для забезпечення можливості коректного порівняння двох підходів до автомасштабування використано однорідне навантаження (генерація PDF-документів). Такий вибір дозволяє ізолювати ефект саме автомасштабування від впливу варіативності завдань. Дослідження ефективності фреймворку при гетерогенному навантаженні є перспективним напрямом подальших досліджень.

Недоліками запропонованого фреймворку є декілька зовнішніх факторів, які можуть мати негативний вплив на ефективність автомасштабування. Так, функції AWS Lambda мають обмеження по часу

виконання, що робить їх непридатними для довготривалих операцій обробки даних. Застосування напівмарковських моделей може призводити до підвищення обчислювальної складності і додаткових витрат. Кількість станів моделі впливає на точність прогнозу та потребує ретельного налаштування [103].

Це передбачає подальшу еволюцію фреймворку та його адаптацію для роботи з нерівномірно розподіленими завданнями. Також варто дослідити можливість пріоритезації завдань на основі їх розміру та часу виконання. Це дозволило б зменшити час очікування та зробити додаток більш стабільним.

Результати порівняння двох підходів до прогнозного автоматичного масштабування безсерверних обчислювальних ресурсів, а саме на основі нейронних мереж (DeepAR) та на основі аналітичної моделі з напівмарковськими процесами наведено у таблиці 4.2.

Таблиця 4.2.

Зведення результатів двох серій експериментів з прогнозного автомасштабування

Характеристика	DeepAR	Аналітична модель
Набір даних	1082178 подій	1025132 записи
Зменшення холодних стартів	~27% від базового рівня	до 3% залишкових
Прискорення обробки	не входило до цілей експерименту	+25,8%
Збільшення пропускної здатності	не входило до цілей експерименту	+21,3%
Зменшення кількості необроблених запитів	~14%	визначається формулою (3.20)
Потреба в історичних даних	так (≥ 30 кроків)	ні
Аналітичні гарантії SLA	ні (евристичні пороги)	так (формула (3.20))
Адаптація до змін навантаження	перенавчання моделі	Крок 8 (переоцінка параметрів)
Обчислювальна вартість	висока (GPU для навчання)	низька (аналітичні формули)

Оскільки експерименти проводилися на різних наборах даних та у різний час, наведена таблиця має характер зведення результатів, а не контрольованого порівняння. Пряме кількісне зіставлення окремих метрик між підходами є некоректним; таблиця дозволяє лише якісно оцінити спектр переваг та обмежень кожного підходу.

Аналіз оцінки результатів зведення свідчить про перевагу аналітичного підходу за більшістю показників. Так, однією з відмінностей аналітичного підходу є наявність аналітичних гарантій якості обслуговування. Формула для оцінки кількості відхилених завдань (3.20) дозволяє визначити оптимальне значення provisioned concurrency, що забезпечує задане SLA (наприклад, $\theta(T) \leq 0{,}001$ для SLA 99,9%). Підхід на основі DeepAR в свою чергу використовує евристичні порогові значення t_{up} та t_{down} (Algorithm 2), які підбираються емпірично і не забезпечують аналітичних гарантій. Згідно з результатами Розділу 3, ці евристичні пороги можуть бути замінені відповідними значеннями на основі формул (3.15), (3.16).

4.3 Імітаційне моделювання СМО з неоднорідним потоком

Для верифікації теоретичних результатів Розділу 3 здійснювалося імітаційне моделювання. Метою імітаційного моделювання є підтвердження відмінності між моделлю суміші потоків (3.2) та MMPP-наближенням (підпункт 3.1.3), переваги оптимального протоколу вибору серверу над випадковим (підпункт 3.3.2), перевірки адекватності нормального наближення для розподілу довжини черги (Теорема 3.2.3).

Для проведення імітаційного моделювання розроблено програмний модуль мовою Python із використанням бібліотек NumPy [176], SciPy [177], Matplotlib [178] та Scikit-learn [179]. Модуль реалізує повний конвеєр обробки даних, а саме завантаження даних, розбиття на інтервали, оцінка параметрів суміші, статистичне тестування, побудова функцій інтенсивності, моделювання методом Монте-Карло [180–182] та генерація звітів.

В якості вхідних даних використано реальні метрики функціонування безсерверної системи, розгорнутої на платформі AWS. Набір даних включає 578 спостережень, що характеризують моменти надходження та тривалість обробки завдань, зібраних протягом 48 годин (172 800 с). Статистичний аналіз показав, що середній час обробки становить 0,241 с (медіанне значення – 0,252 с), при мінімальному значенні 0,016 с та максимальному – 0,615 с; стандартне відхилення дорівнює 0,102 с. Середня інтенсивність надходження завдань оцінюється на рівні 0,00335 задач/с (еквівалентно 0,201 задач/хв), тоді як середній інтервал між послідовними надходженнями складає 299,5 с.

Параметри імітаційної моделі обрано наступним чином:

- кількість каналів обслуговування становить c=3 з пропускною здатністю $\mu_1 = 0{,}03$, $\mu_2 = 0{,}06$, $\mu_3 = 0{,}09$ задач/с;
- максимальний розмір черги обмежений значенням $N_L = 10$ (завдання, що надходять при наповненій черзі, відхиляються);
- інтенсивність вхідного потоку масштабується за допомогою коефіцієнта 189,0, що дозволяє забезпечити завантаженість системи на рівні ρ≈85%.

Зауважимо, що такі параметри імітаційної моделі (c=3, $N_L = 10$) обрано з урахуванням двох міркувань. По-перше, мала кількість серверів дозволяє чітко виявити відмінності між протоколами маршрутизації та моделями вхідного потоку, які при великій кількості серверів нівелюються ефектом усереднення. По-друге, аналітичні результати Розділу 3 сформульовані для довільних c та N_L, і імітаційне моделювання спрямоване на перевірку якісних залежностей (оптимальний протокол кращий за випадковий, суміш потоків краща за MMPP), а не на відтворення абсолютних значень метрик конкретної виробничої системи.

Для отримання статистично надійних результатів проведено 100 прогонів за методом Монте-Карло.

Розглядалися дві архітектури вхідного процесу:

- Система 1 – однопотокова модель з неоднорідним пуассонівським процесом із інтенсивністю $\lambda(t) = \sum_{i=1}^{k} w_i\,(t)\lambda_i(t)$. Генерація подій здійснюється алгоритмом проріджування (thinning) [183], де спочатку генерується пуассонівський потік з максимальною інтенсивністю $\lambda_{\max}$, потім кожна подія приймається з ймовірністю $\lambda(t)/\lambda_{\max}$. Ця модель відповідає MMPP-наближенню;
- Система 2 – багатопотокова модель $N(t) = \sum_{i=1}^{k} w_i\,(t)N_i(t)$, де k незалежних пуассонівських процесів генеруються окремо і об'єднуються із сортуванням за часом. Ця модель відповідає запропонованій моделі суміші (3.2).

Для кожної архітектури порівнювалися два протоколи вибору серверу, а саме випадковий (вибір довільного вільного сервера з рівномірним розподілом) та оптимальний (вибір найпотужнішого вільного сервера з максимальним μ_i).

4.3.1 Оцінка параметрів суміші на реальних даних

На першому етапі дослідження часовий ряд розбито на 19 інтервалів тривалістю 8400 с кожний (за винятком останнього інтервалу тривалістю 10800 с), із забезпеченням мінімальної кількості спостережень на рівні не менше 30 задач у кожному інтервалі. Для кожного інтервалу здійснено оцінювання параметрів моделі суміші нормальних розподілів (Gaussian Mixture Model)[184] із застосуванням EM-алгоритму [185], при цьому оптимальна кількість компонентів $K \in \{1, 2, 3, 4, 5\}$ визначалась на основі байєсівського інформаційного критерію (BIC) [186].

Отримані результати (рис. 4.10) свідчать про виражену нестаціонарність вхідного процесу. Зокрема, кількість компонентів суміші $K(t)$ варіюється від 1 (однорідний режим навантаження, зафіксований у 9 інтервалах) до 4 (складна багатокомпонентна структура – у 2 інтервалах), тоді як значення $K = 2$ спостерігається у 8 інтервалах. Перехід між різними значеннями K відбувається у 11 з 18 випадків (61,1%), що вказує на значну

зміну структури навантаження в часі. Крім того, середні значення часу обробки $\mu_i(t)$ у компонентах суміші варіюються від 0,020 с (швидка обробка невеликих завдань) до 0,560 с (повільна обробка складних завдань), що відрізняється у 28 разів. Також ваги компонентів $w_i(t)$ змінюються від 0,033 (мінорний компонент, 3,3% трафіку) до 1,0 (єдиний компонент), що відображає суттєву зміну пропорцій між типами навантаження.

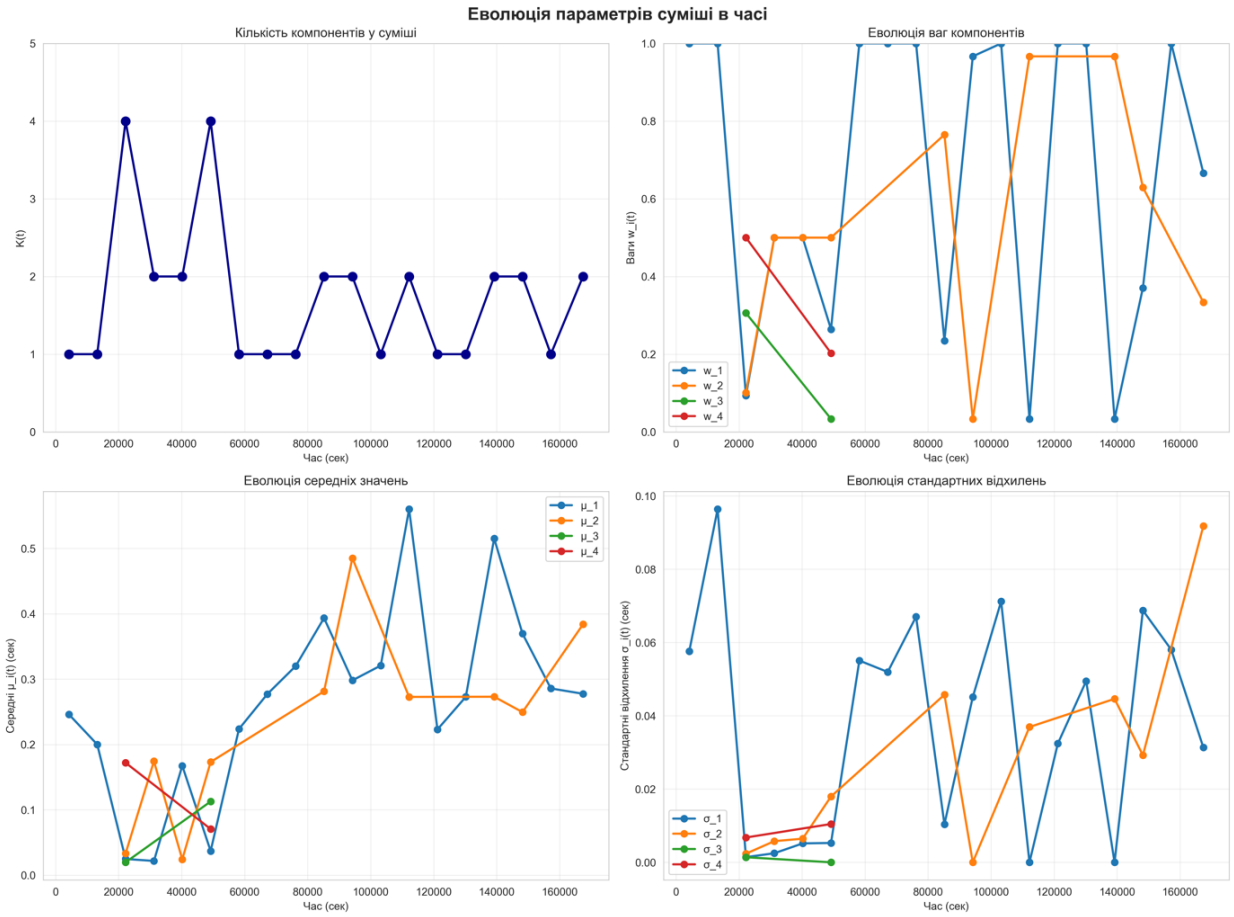

Рис. 4.10 – Еволюція параметрів суміші в часі: $K(t)$, $w_i(t)$, $\mu_i(t)$, $\sigma_i(t)$

На рис. 4.11 показано вертикальні розподіли сумішей на кожному з 19 часових інтервалів. Візуалізація демонструє, що на ранніх інтервалах (0–45000 с) переважають розподіли з двома чітко вираженими кластерами (швидкі та повільні завдання), тоді як на пізніх інтервалах (90000–172800 с) розподіл стає більш однорідним з одним домінуючим компонентом.

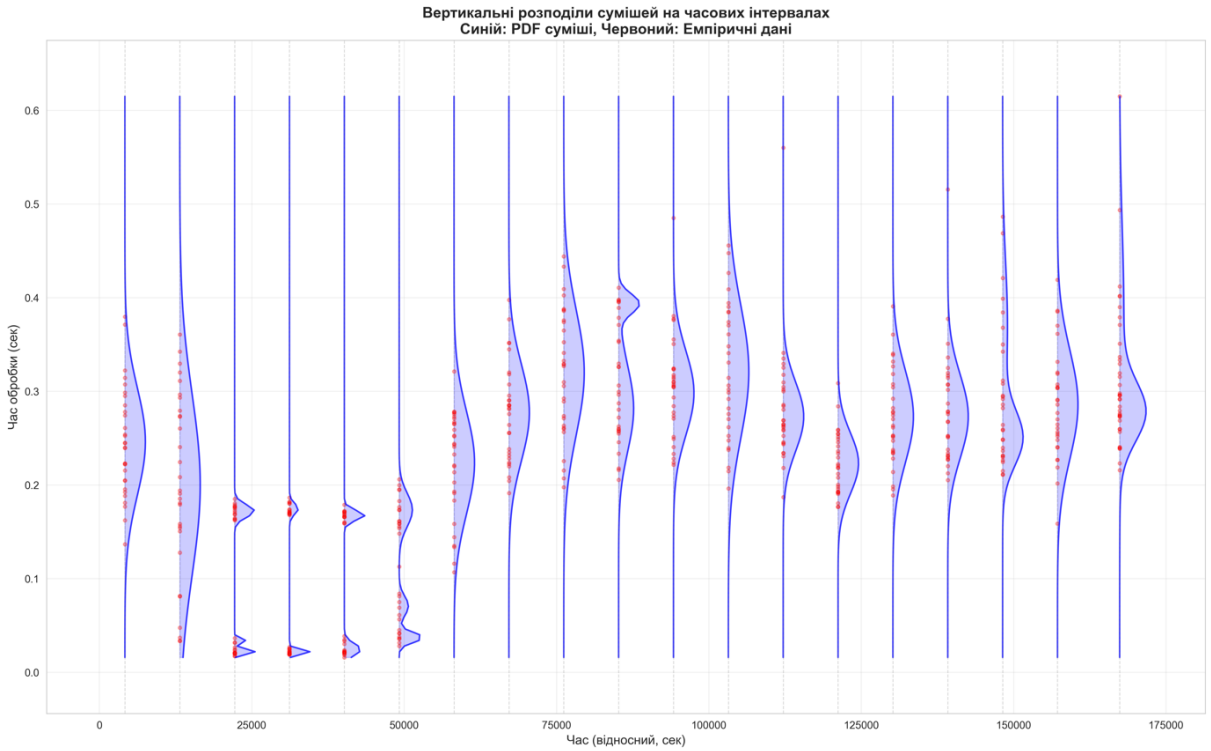

Рис. 4.11 – Вертикальні розподіли сумішей на часових інтервалах

Для кількісної оцінки нестаціонарності проведено тести Колмогорова–Смірнова [187, 188] для 18 пар послідовних інтервалів (рис. 4.12). Верхня частина рисунку демонструє D-статистику (відстань між емпіричними функціями розподілу) з критичним значенням, нижня частина – p-value з порогом α=0,05. Пари інтервалів, для яких нульову гіпотезу про статистичну однорідність розподілів відхилено, виділено червоним кольором, що дозволяє ідентифікувати часові ділянки зі статистично значущими змінами характеристик вхідного потоку.

Рис. 4.12 – Результати тестів Колмогорова–Смірнова: D-статистика

Отримані результати демонструють, що у 8 з 18 пар (44,4%) розподіли є статистично різними ($p < 0{,}05$). Найбільші відмінності спостерігаються при переходах між інтервалами 1–2, 5–6, 6–7, 7–8 та 12–13, що збігається з моментами зміни кількості компонентів K. Це обґрунтовує використання неоднорідного вхідного процесу $H_k(t)$ у теоретичній моделі.

4.3.2 Алгоритм налаштування параметрів СМО

Як зазначалося в третьому розділі важливим аспектом для визначення оптимальної кількості серверів та їх потужності, є алгоритмічний підхід до визначення кількості серверів (кластерів обробки) та коректного об’єднання вхідних потоків у конкретні підпотоки для більш оптимального розподілу навантаження між серверами та зменшення часу очікування (тобто вартості) для оптимального розподілу завдань. Слід зауважити, що всі розроблені методи Розділу 3 є статистичними, а отже, вимагають достатньої кількості завдань для оцінки параметрів вхідних потоків, а саме інтенсивностей $\lambda_i(t)$

та ваг розподілу $w_i(t)$ для процесу надходження завдань. Оскільки граничний процес згідно з Теоремою 3.2.3. є розв'язком стохастичного диференціального рівняння Ito, то для аналізу можна використовувати оцінки нормального розподілу. Використовуючи такі оцінки, можна стверджувати що величина вибірки при дослідженні може не перевищувати $N_{obs,\Delta} \geq 50$, де параметр Δ визначає довжину інтервалу, на якому буде зібрана інформація. Варто зазначити, що даний інтервал уже введений у Теоремі 3.2.3. для більш чіткого та цілісного розуміння контексту збору інформації. В контексті AWS Lambda даний параметр відповідає інтервалу моніторингу CloudWatch (від 1 с для детального моніторингу до 60 с для базового).

Розглянемо основний алгоритм дослідження системи та визначення її параметрів на основі кроків, розглянутих у Розділі 3. Для цього визначимо цільову функцію, яка може бути розглянута в якості функції для оптимізації. Як було зазначено вище, одним із основних параметрів для оптимізації є вартість користування серверам із потрібною потужністю, тобто $T_{cost}(S, R)$, де $T_{cost}(S, R)$ – загальна вартість обслуговування (резервування) серверів за одиницю часу, S - резервних серверів та активних серверів R. В AWS Lambda це відповідає сумарній вартості provisioned concurrency, що відстежується через метрики AWS Cost Explorer.

Другим важливим показником при аналізі вищевказаної задачі є кількість відхилених завдань (відхилених запитів) $\theta(T)$ за період T за рахунок неспроможності обчислень на основі наявної кількості активних серверів чи загальної кількості серверів. В AWS Lambda це відповідає метриці Throttles у CloudWatch. Слід відзначити, що даний параметр є критичним у випадку надсилання пакетів або підзавдань на виконання, і порушення одного із кроків може призвести до порушення розв'язання задачі в цілому.

Третім важливим показником будемо вважати час «простою» сереверів, або відповідно до першого показника – кількість неактивних

серверів, причому час «простою» будемо позначати через D_T за час T. В AWS Lambda це відповідає часу, протягом якого provisioned concurrency не обробляє жодного запиту, що призводить до витрат без корисного навантаження.

Ці три основні показники з одного боку можна розглядати як багатокритеріальну задачу, тобто розглядати наступну задачу

$$\begin{cases} T_{cost}(S,R) \to MIN, \\ \theta(T) \to MIN, \\ D_T \to MIN. \end{cases}$$

З іншого боку, один із цих факторів можна трактувати як основну цільову функцію, а інші два як додаткові умови, які будуть задовольняти деяким наперед заданим критеріям. Наприклад, відповідну оптимізаційну задачу можна математично описати наступним чином

$$T_{cost}(S,R) \to MIN,$$

за умови

$$\begin{cases} \theta(T) \le \theta_{crit}, \\ D_T \to D_{crit}. \end{cases}$$

У першому випадку умова є багатокритеріальною і відповідно може бути розв’язана методами багатокритеріального програмування або використовуючи метаевристичні алгоритми. Друга однокритеріальна задача також може бути розв’язана з використанням метаевристичних алгоритмів із врахуванням умов на частку відмов $\theta(T)$ та часу простою D_T. Проте слід відмітити, що метаевристичні алгоритми не можуть претендувати на алгоритми що знаходять глобальний оптимум, а можуть бути розглянуті лише у якості хорошого початкового наближення для подальших більш точних алгоритмів.

Проте для досягнення більшої точності та оптимізації вартості оренди ресурсів використовуємо алгоритм параметричної оцінки конфігурації безсерверної системи, який наведений у підпункті 3.3.5 Розділу 3. Як було зазначено вище, на *початковому кроці* слід одержати достатню кількість

оброблених запитів для оцінки параметрів СМО. Дана кількість може бути обмежена одним з двох критеріїв:

– або кількістю, необхідною для перевірки нормальності розподілу приросту процесу, що визначається стохастичним диференціальним рівнянням Ітo

$$d\check{L}^0(t) = C_1\left(\check{L}^0(t), \rho(t), t\right)dt + \sqrt{\frac{1}{2}B(\rho(t), t)}dW(t)$$

згідно Теореми 3.2.3,

– або необхідною точністю статистичної перевірки для порівняння сумішей на кожному кроці.

Коли необхідна початкова кількість вхідних даних отримана, то переходимо до *кроку 2*, на якому здійснюється одночасно ідентифікація джерел та збір даних для подальших кроків порівняння, при цьому будемо вважати, що параметри системи будуть переглядатися або через фіксовані проміжки часу T_{check} або через певну кількість оброблених задач N_{check}. Припустимо, що здійнена початкова оцінка параметрів системи $\left(k(t), w_i(t), \lambda_i(t)\right)$ у момент часу t. Зауважимо, що оцінка буде здійснена на основі методу максимальної правдоподібності, описаного в Розділі 3. Під час здійснення оцінювання на інтервалах часу буде одержано наступні характеристики сумішей розподілу (рис. 4.11). Як ми можемо бачити з даного рисунку, розподіл часу обробки завдань буде сильно коливатися між сусідніми інтервалами, проте кумулятивний розподіл суміші розподілу буде збігатися до певного стаціонарного розподілу на достатньо довгому інтервалі часу, якщо система залишаються однорідною, тобто параметри $\left(k(t), w_i(t), \lambda_i(t)\right)$ не будуть змінюватися з часом.

У даному випадку підбирання кількості та потужності серверів буде здійснюватися на основі підходу, запропонованого нами в підпункті 3.3.4, в якому зроблено припущення що інтенсивність надходження повинна бути пропорційна інтенсивності обробки завдань. Для перевірки потреби зміни

конфігурації системи масового обслуговування може бути використаний тест Колмогорова–Смірнова, причому даний тест є непараметричним, а отже буде мати меншу потужність у порівняння із параметричними тестами. Для того, щоб пересвідчитися у коректності тесту можна також користатися і відстанями між ймовірнісними мірами, які можуть давати дещо наочний вигляд зміни розподілу вхідного процесу, а саме відстанями Геллінґера [189], відстанню повної варіації [190], відстанню Леві [191] тощо. Дані метрики дозволяють відслідкувати зміну відстані у числовому вимірі, що на відміну від тесту Колмогорова–Смірнова може бути навіть ефективніше.

Таким чином, остаточний алгоритм налаштування параметрів системи масового обслуговування можна описати наступним чином із врахуванням трьох цільових функцій, описаних вище.

Algorithm 3. Налаштування параметрів СМО

1: Задання параметрів системи $N_{obs,\Delta}, T_{check}, N_{check}$

2: Збір даних для початкової оцінки параметрів $N_{obs,\Delta}$

3: Оцінка початкових параметрів $\left(\hat{k}(t), \hat{w}_i(t), \hat{\lambda}_i(t),\ i = 1, \dots, \hat{k}(t)\right)$ на основі методу максимальної правдоподібності та ЕМ алгоритму

4: Налаштування кількості серверів $K(t)$ та інтенсивностей серверів $\mu_i,\ i = 1, \dots, K(t)$ на основі параметрів $\left(\hat{k}(t), \hat{w}_i(t), \hat{\lambda}_i(t), i = = 1, \dots, \hat{k}(t)\right)$ з врахуванням додаткових обмежень

5: Збір інформації для кроку порівняння – час T_{check} або кількість нових завдань N_{check}

6: Переобчислення нових параметрів суміші $\left(\hat{k}^{new}(t), \hat{w}_i^{new}(t), \hat{\lambda}_i^{new}(t), i = 1, \dots, \hat{k}^{new}(t)\right)$ для $N_{obs,\Delta} + +N_{check}$ оброблених завдань

7: $N_{obs,\Delta} = N_{obs,\Delta} + N_{check}$

8: Порівняння розподілів, заданих параметрами $\left(\hat{k}(t), \hat{w}_i(t), \hat{\lambda}_i(t), i = 1, \dots, \hat{k}(t)\right)$ та $\left(\hat{k}^{new}(t), \hat{w}_i^{new}(t), \hat{\lambda}_i^{new}(t), i = 1, \dots, \hat{k}^{new}(t)\right)$ на основі тесту Колмогорова – Смірнова або відстаней між розподілами

9: if час роботи системи не завершено then

10: if розподіли відрізняються then

11: $\left(\hat{k}(t), \hat{w}_i(t), \hat{\lambda}_i(t), i = 1, \dots, \hat{k}(t)\right) =$

$=\left(\hat{k}^{new}(t), \hat{w}_i^{new}(t), \hat{\lambda}_i^{new}(t), i = 1, \dots, \hat{k}^{new}(t)\right)$

Переходимо до кроку 4

else

Переходимо до кроку 5

end if

else

Вихід

end

Algorithm 3 є деталізацією процедури адаптивного моніторингу (Крок 8 алгоритму параметричної оцінки з підпункту 3.3.5) та об'єднує кроки статистичної оцінки параметрів, перевірки стаціонарності та реконфігурації системи. Його коректність забезпечується збіжністю EM-алгоритму оцінки параметрів суміші (крок 3) та консистентністю тесту Колмогорова–Смірнова (крок 8): при зростанні обсягу вибірки $N_{obs,\Delta}$ оцінки параметрів наближаються до істинних значень, а потужність тесту зростає.

Параметри T_{check} та N_{check} визначаються на основі компромісу між чутливістю до змін навантаження та обчислювальними витратами на переоцінку. У контексті AWS Lambda нижня межа T_{check} обмежена інтервалом моніторингу CloudWatch (від 1 с для детального до 60 с для базового моніторингу), а $N_{check} \geq 50$ забезпечує мінімальний обсяг вибірки для коректності нормального наближення. Детальне дослідження чутливості показників системи до варіювання цих параметрів є напрямом подальших досліджень.

У випадку, коли структура навантаження змінюється швидше за інтервал T_{check}, алгоритм виявить зміну із затримкою в один цикл моніторингу. Для мінімізації такої затримки можна використовувати подвійний критерій: зменшений T_{check} у поєднанні з метриками відстані між розподілами (Геллінґера, повної варіації), які, на відміну від тесту Колмогорова–Смірнова, дозволяють кількісно відстежувати ступінь зміни розподілу і реагувати до досягнення порогу статистичної значущості.

Практична верифікація основних кроків Algorithm 3 здійснена у рамках другого експерименту (підпункт 4.2). Зокрема, кроки 2–3 реалізовані через збір метрик CloudWatch та оцінку параметрів вхідного потоку, крок 4 відповідає налаштуванню provisioned concurrency за допомогою Algorithm 1 (підпункт 2.4), а кроки 5–8 реалізовані через періодичний моніторинг та переоцінку параметрів з інтервалом N хвилин. Імітаційне моделювання (підпункт 4.3) додатково верифікує крок 8 – порівняння розподілів за тестом Колмогорова–Смірнова – та демонструє обґрунтованість використання цього тесту для прийняття рішення про реконфігурацію.

4.3.3 Результати імітаційного моделювання методом Монте-Карло

Імітаційне моделювання виконано для чотирьох конфігурацій системи ({Система 1, Система 2} × {Random, Optimal}), по 100 прогонів для кожної конфігурації. У кожному прогоні генерувалося в середньому 109 200 заявок (за рахунок використання множника інтенсивності 189,0), з яких близько 107100 оброблялися успішно, тоді як приблизно 2080 заявок відхилялися. Відповідно, середній рівень пропускної здатності системи становить близько 98,1%. Зведені результати наведено у таблиці 4.3.

Таблиця 4.3.

Результати моделювання методом Монте-Карло

Конфі-гурація	Error Rate	Сер. довжина черги	Сер. час очікування (с)	Надійшло	Оброб-лено	Від-хиле-но
System 1 + Random	0,01929 ± 0,00097	3,2950 ± 0,0576	4,4705 ± 0,0773	109 270	107 157	2107
System 1 + Optimal	0,01901 ± 0,00100	3,2452 ± 0,0585	4,4022 ± 0,0785	109 270	107 188	2077
System 2 + Random	0,01911 ± 0,00114	3,2873 ± 0,0572	4,4603 ± 0,0761	109 197	107 104	2087
System 2 + Optimal	0,01881 ± 0,00118	3,2395 ± 0,0647	4,3947 ± 0,0871	109 197	107 137	2055

Порівняльний аналіз протоколів вибору сервера показав, що оптимальний протокол, який передбачає призначення заявки на найпотужніший вільний сервер, забезпечує статистично значуще зниження рівня відмов порівняно з випадковим вибором. Зокрема, середнє значення Error Rate становить 0,01891 для оптимального підходу проти 0,01920 для випадкового, що відповідає покращенню на 1,51% ($p = 0{,}0104$ за t-критерієм Стьюдента [192–194]). Покращення спостерігається також за іншими показниками ефективності системи. Середня довжина черги зменшується на 1,54% (з 3,2912 до 3,2424), а середній час очікування – на 1,50% (з 4,4654 до 4,3985). На рис. 4.13 наведено порівняння двох протоколів за трьома основними метриками з урахуванням довірчих інтервалів.

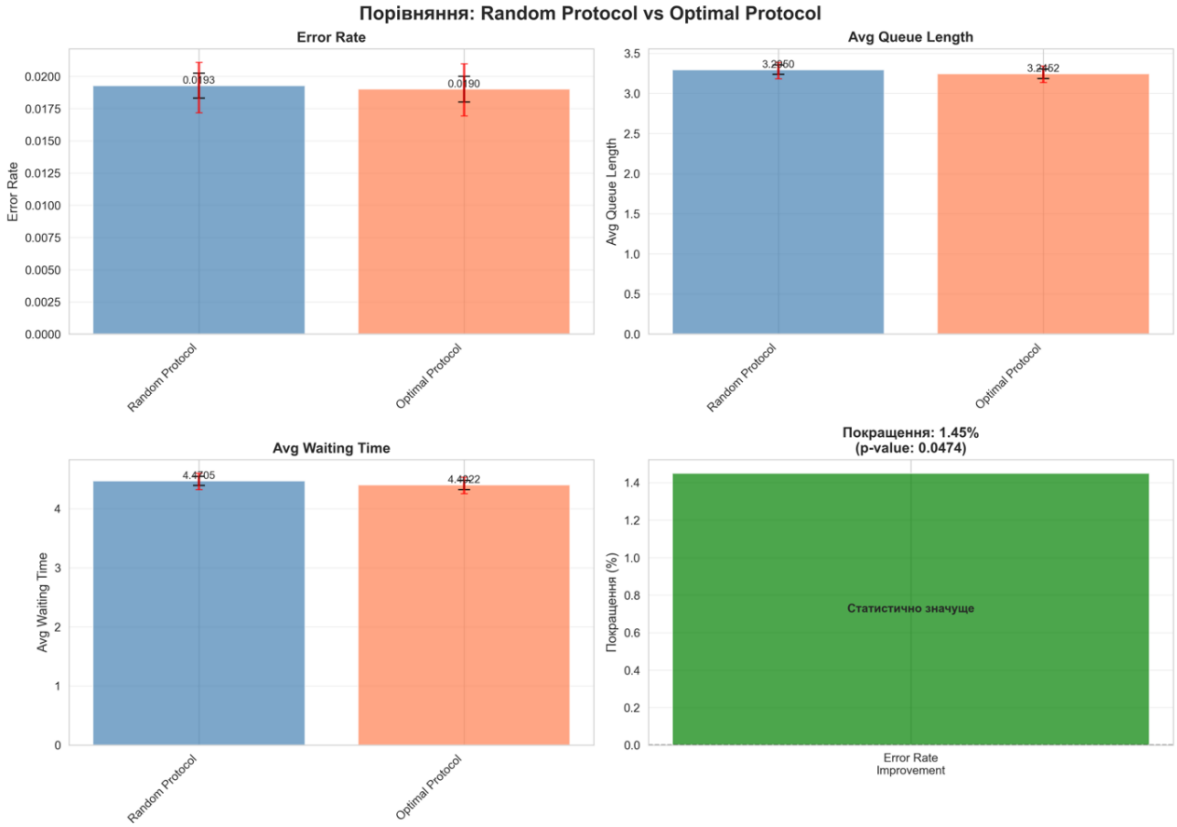

Рис. 4.13 – Порівняння Random та Optimal протоколів

Даний результат підтверджує теоретичний висновок з підпункту 3.3.4, де аналітично доведено $L_q^{\text{opt}} < L_q$ та $W_q^{\text{opt}} < W_q$ для *M/M/2* системи з неоднорідними серверами, що підтверджує доцільність використання

оптимального протоколу маршрутизації заявок. Зауважимо, що Система 2 (Multi-stream, модель суміші (3.2)) демонструє дещо кращі показники ефективності порівняно із Системою 1 (Single-stream, MMPP). Зокрема, середнє значення Error Rate для Системи 2 становить 0,01896 проти 0,01915 для Системи 1, що відповідає покращенню на 0,95%. Аналогічна тенденція спостерігається і для інших метрик: середня довжина черги зменшується з 3,2701 до 3,2634, а середній час очікування – з 4,4364 до 4,4275. Різниця між Системою 1 та Системою 2 за Error Rate не досягає статистичної значущості ($p = 0{,}2260$) при рівні завантаженості $\rho \approx 85\%$, що пояснюється відносно малим внеском дисперсійної компоненти при даному ρ.

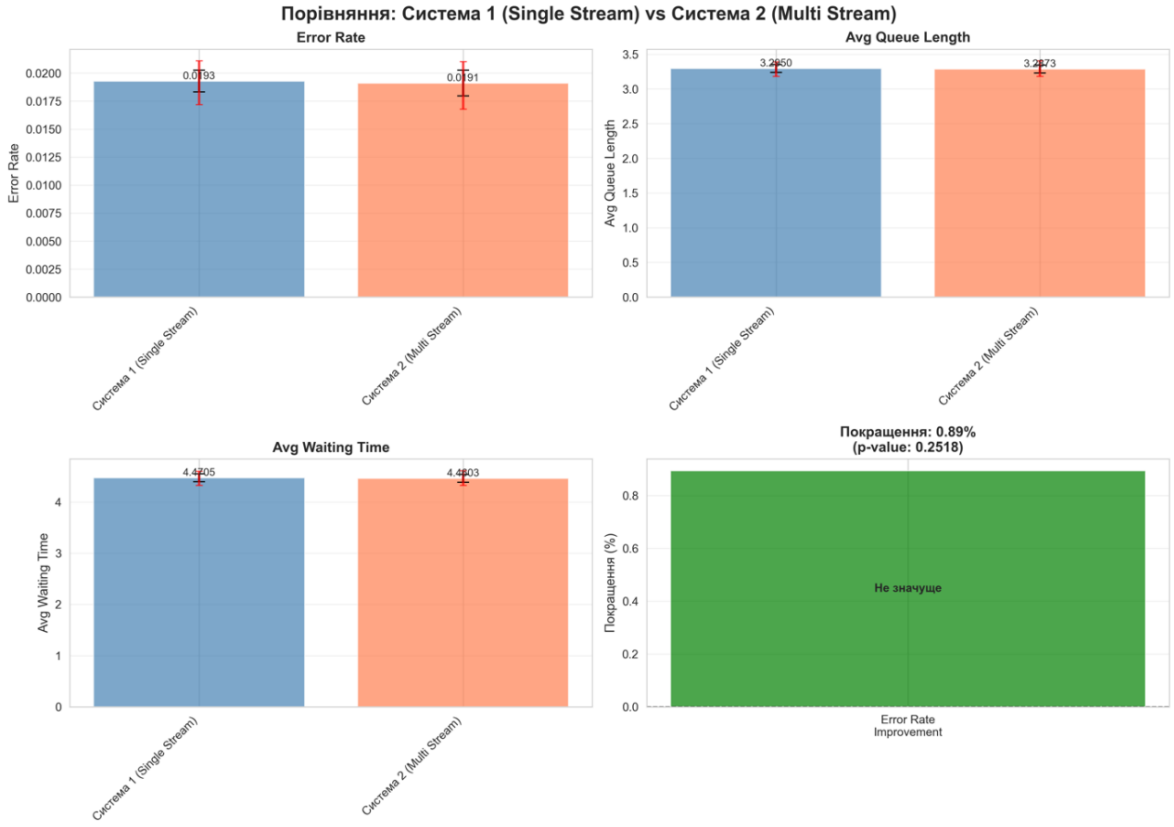

Рис. 4.14 – Порівняння Системи 1 (Single Stream) та Системи 2 (Multi Stream)

Разом з тим, у всіх 100 імітаційних прогонах Система 2 демонструє менше середнє значення Error Rate (рис. 4.14). Отриманий результат узгоджується з теоретичними положеннями підпункту 3.1.3, оскільки $\mathrm{Var}\big(N(t)\big) \leq \mathrm{Var}\left(\widetilde{N}(t)\right)$, тобто модель суміші має меншу дисперсію вхідного

процесу, що призводить до меншої ймовірності переповнення черги і, відповідно, меншого Error Rate. При вищому рівні завантаженості ($\rho \to 1$) різниця між моделями набуватиме більш вираженого характеру, підсилюючи переваги використання багатопотокової моделі суміші, що є напрямом подальших досліджень.

Найбільш ефективною виявилась конфігурація System 2 + Optimal, яка поєднує модель суміші потоків із оптимальним протоколом вибору сервера. Для неї отримано такі характеристики: Error Rate $= 0{,}01881 \pm 0{,}00118$, 95% довірчий інтервал становить $[0{,}01655; 0{,}02086]$, а середня кількість відхилених заявок – 2055 із 109197, що відповідає пропускній здатності 98,12%.

Аналіз розподілу Error Rate за 100 імітаційними прогонами (рис. 4.15) показує, що для всіх чотирьох конфігурацій гістограми мають форму, близьку до нормального розподілу. Це узгоджується з дифузійною апроксимацією (Теорема 3.2.3), згідно з якою при великій інтенсивності потоку заявок процес довжини черги $L(t)$ може бути апроксимований процесом процес Орнштейна–Уленбека, стаціонарний розподіл якого наближається до нормального. Серед усіх розглянутих конфігурацій саме System 2 + Optimal характеризується найменшим середнім значенням Error Rate ($\mu = 0{,}01881$, $\sigma = 0{,}00118$), що підтверджує доцільність одночасного використання моделі суміші потоків і оптимальної стратегії маршрутизації заявок [153].

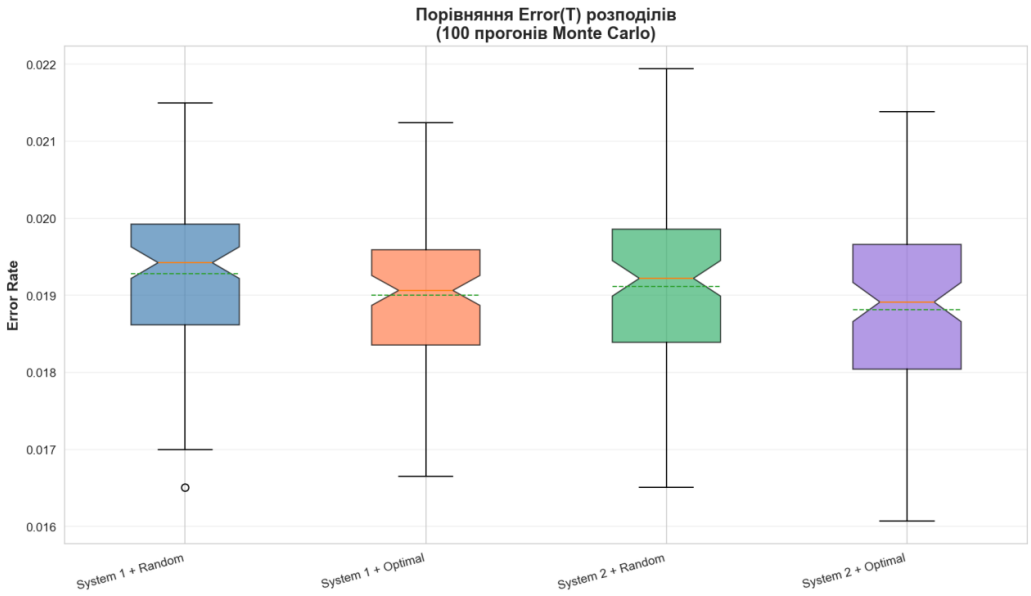

Рис. 4.15 – Розподіл Error(T) за 100 прогонами Монте-Карло для 4 конфігурацій

На рис. 4.16 представлено діаграму розподілів Error(T) для чотирьох досліджуваних конфігурацій. Конфігурація System 2 + Optimal характеризується найнижчою медіаною та мінімальним міжквартильним розмахом, що вказує на вищу стабільність результатів порівняно з іншими конфігураціями при застосуванні оптимального протоколу вибору сервера.

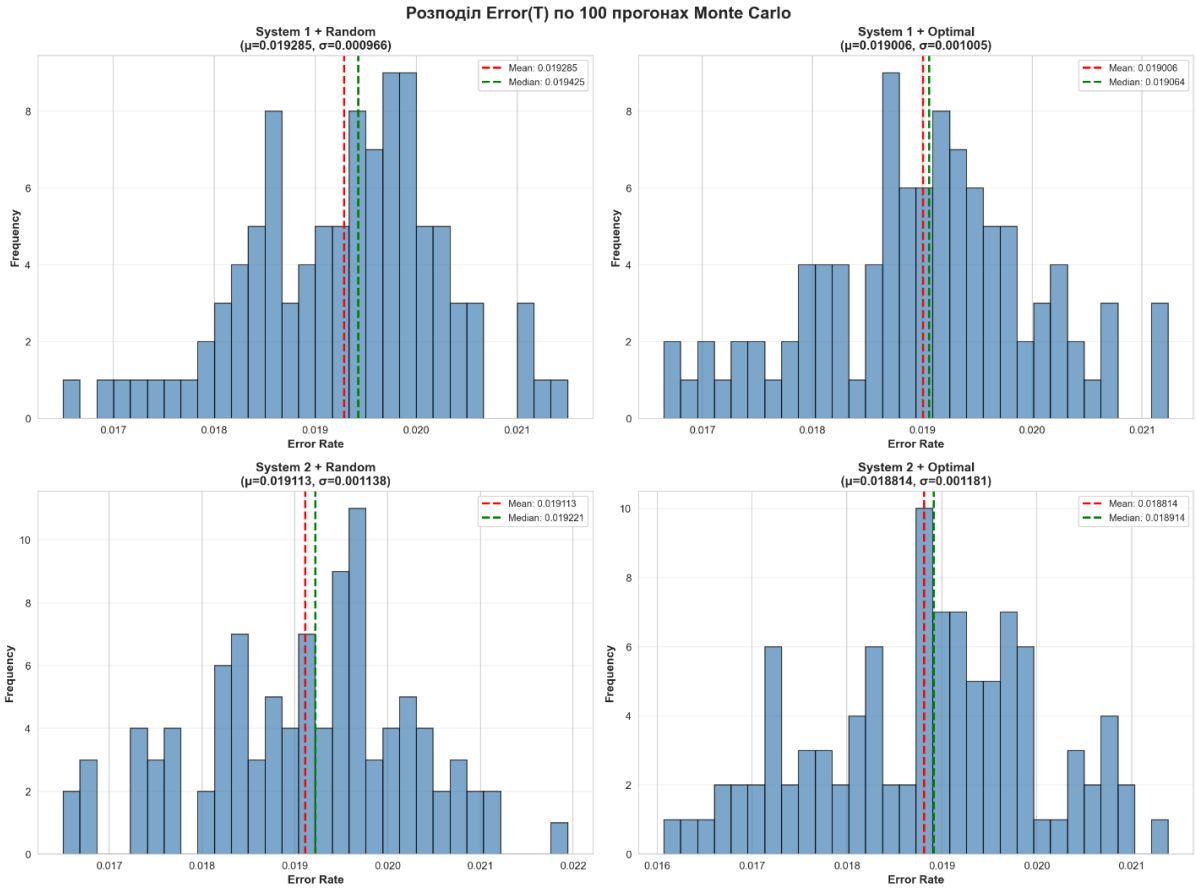

Рис. 4.16 – Порівняння Error(T) для 4 конфігурацій

На рис. 4.17 представлено емпіричні функції розподілу ймовірностей (CDF) для рівня помилок (Error Rate) усіх чотирьох конфігурацій. Крива конфігурації System 2 + Optimal розташована лівіше за інші на всьому діапазоні значень, що свідчить про те, що для будь-якого заданого порогу рівня помилок θ_{SLA} ця конфігурація забезпечує найвищу ймовірність виконання вимог SLA. Отримані результати підтверджують практичну ефективність формули кількості відхилених завдань (throttling rate) (3.20) з Розділу 3.

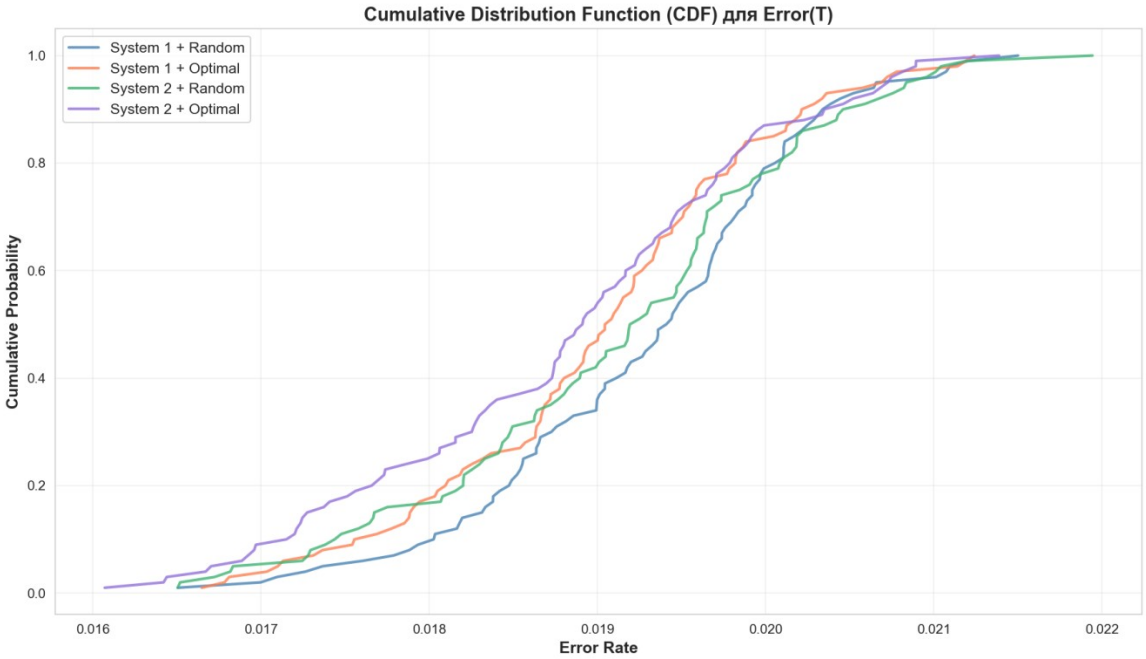

Рис. 4.17 – CDF Error(T) для 4 конфігурацій

Розподіли $p_k = P(L_T = k)$, $k = 0{,}1, \ldots, 10$, для всіх 4 конфігурацій, усереднені по 100 прогонах мають характерну спадну форму (рис. 4.18): максимум досягається при $k = 0$ ($p_0 \approx 0{,}34$-$0{,}35$), потім відбувається різке зменшення до $p_1 \approx 0{,}113$ та подальше плавне спадання до $p_{10} \approx 0{,}018$. Основна відмінність між конфігураціями проявляється при $k = 0$: конфігурації з оптимальним протоколом мають $p_0 \approx 0{,}350$, тоді як конфігурації з випадковим протоколом – $p_0 \approx 0{,}339$. Це означає, що при застосуванні оптимального протоколу система частіше перебуває у стані спокою (порожня черга), що пояснює менший Error Rate: при $p_0 > p'_0$ менша частка часу припадає на стан максимального завантаження ($k = N_L = 10$), а отже менша частка заявок відхиляється.

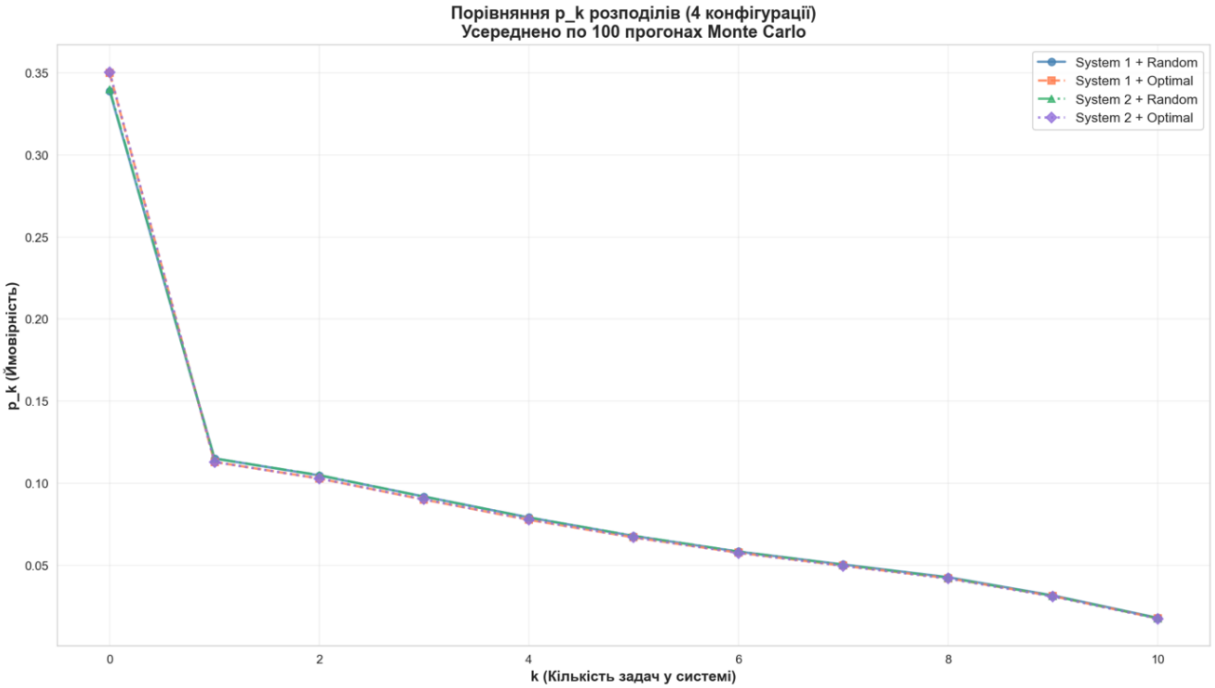

Рис. 4.18 – Порівняння p_k розподілів для 4 конфігурацій, усереднені за 100 прогонами

Окремі p_k розподіли для кожної конфігурації з довірчими інтервалами наведено на рис. 4.19–4.22.

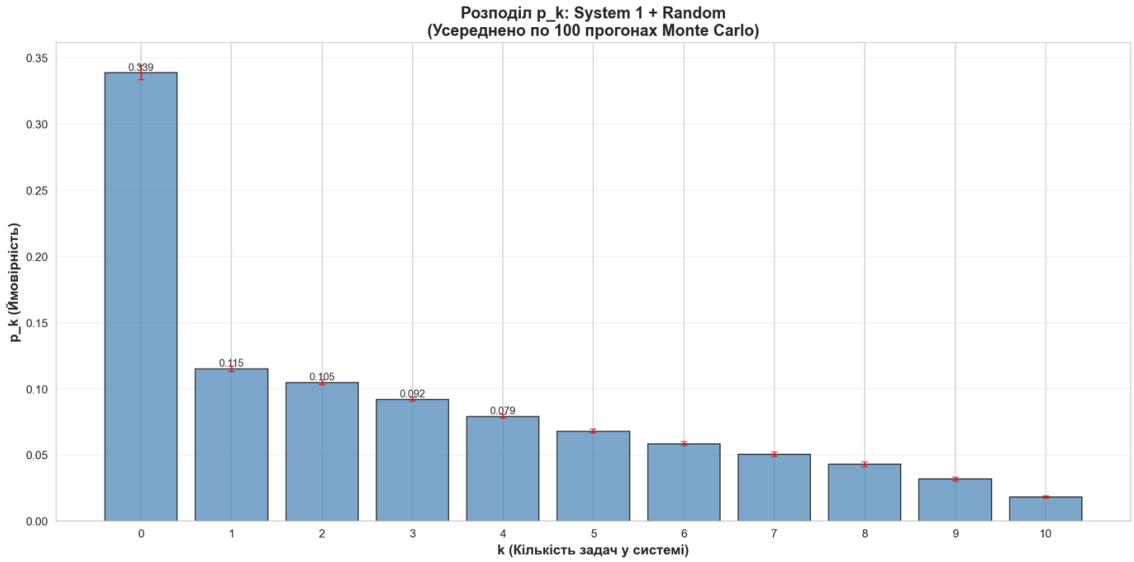

Рис. 4.19 – Розподіл p_k: System 1 + Random

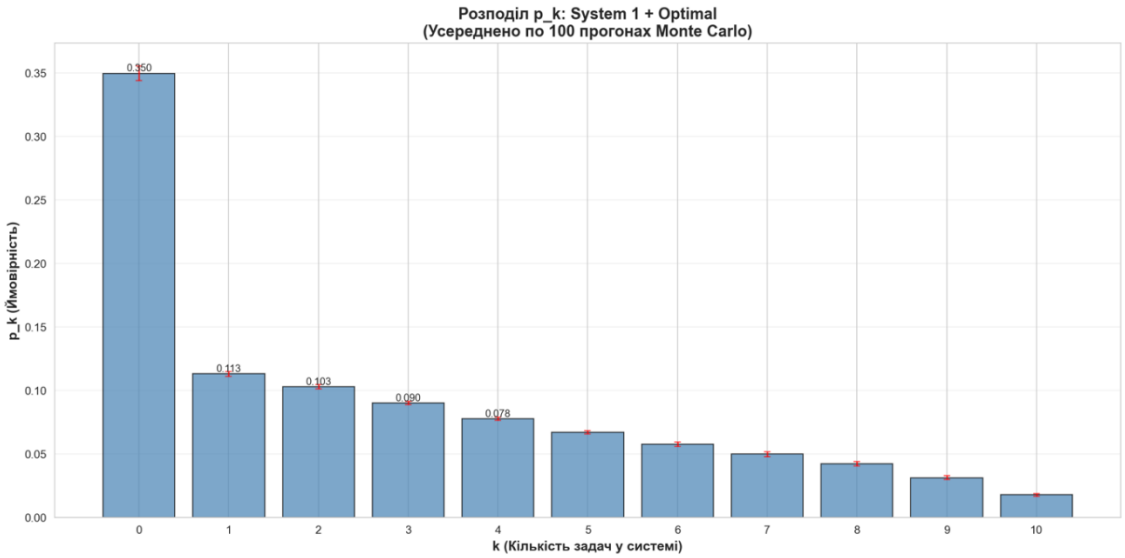

Рис. 4.20 – Розподіл p_k: System 1 + Optimal

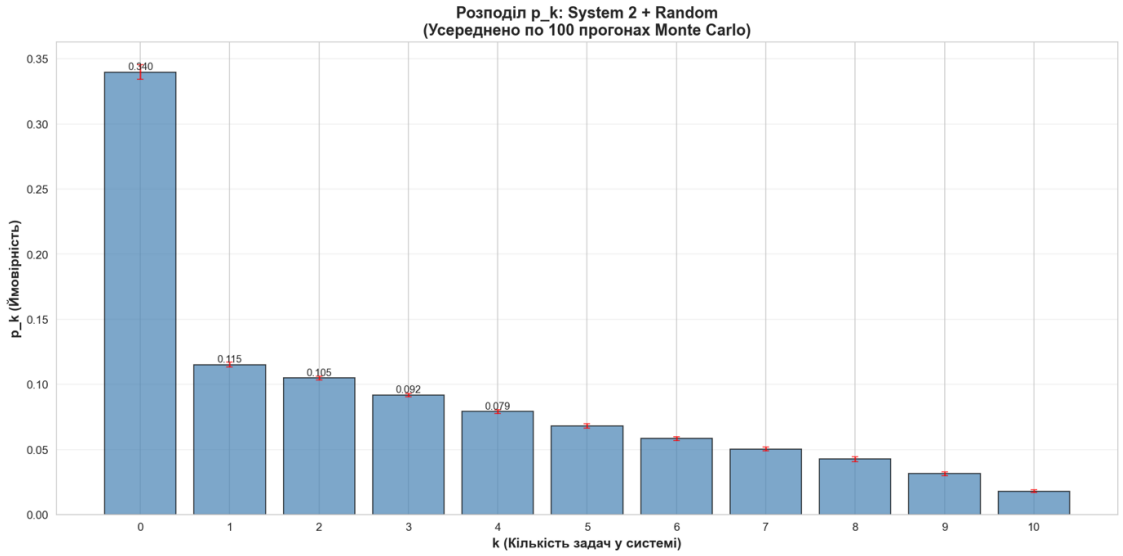

Рис. 4.21 – Розподіл p_k: System 2 + Random

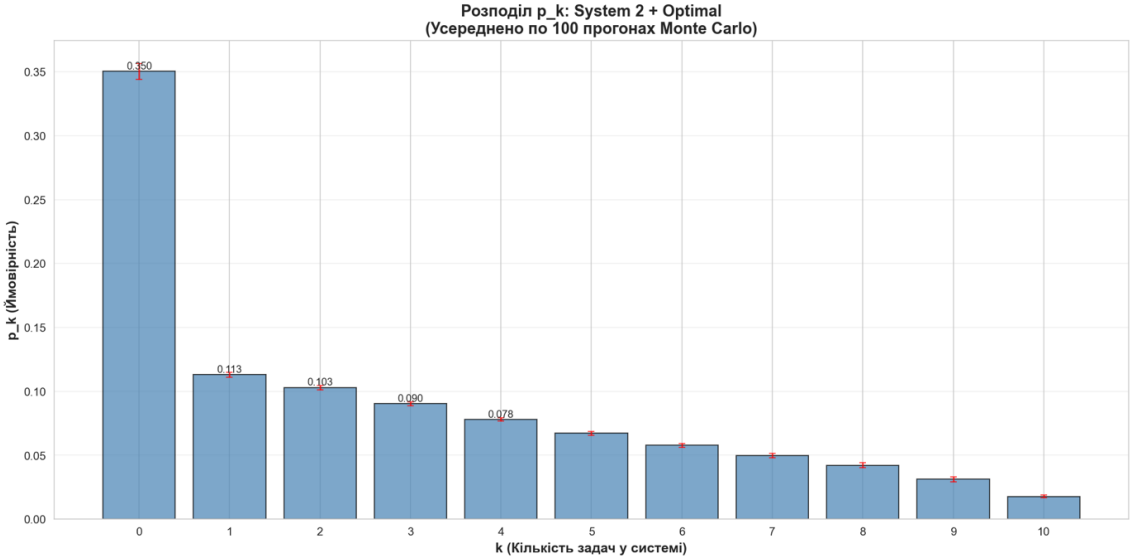

Рис. 4.22 – Розподіл p_k: System 2 + Optimal

Проведені моделювання методом Монте-Карло підтвердили основні теоретичні положення, викладені в Розділі 3, та дозволили кількісно оцінити вплив моделі вхідного потоку і протоколу вибору сервера на ефективність системи. Зокрема, модель суміші (Система 2) систематично забезпечує нижчий рівень помилок (Error Rate) порівняно з MMPP (Система 1), що узгоджується з залежністю між дисперсією вхідного процесу та ймовірністю переповнення черги. Оптимальний протокол вибору сервера статистично значуще покращує показники системи, зменшуючи Error Rate на 1,51%, що підтверджує аналітичні результати для систем з неоднорідними серверами.

Емпіричні розподіли Error Rate наближено нормальні, що підтверджує адекватність дифузійної апроксимації та дозволяє використовувати розрахункові формули для побудови інтервалів надійності та оцінки кількості відхилених завдань. Тести Колмогорова–Смірнова показали значні зміни структури вхідного потоку, що обґрунтовує використання неоднорідної моделі $H_k(t)/M/\infty$ замість класичних стаціонарних M/M/k.

Загалом, результати експерименту демонструють, що застосування оптимального протоколу та моделі суміші підвищує стабільність роботи системи, зменшує частку відхилених заявок і забезпечує ефективне використання ресурсів при змінних умовах навантаження.

Висновки до розділу 4

У четвертому розділі здійснено практичну верифікацію теоретичних результатів дисертаційного дослідження на основі трьох серій експериментів на реальних даних безсерверної інформаційної системи на платформі AWS.

У першому експерименті досліджено прогнозне автомасштабування на основі нейронних мереж DeepAR. На реальних даних (1 082 178 подій за 90 хвилин) прогнозне автоматичне масштабування зменшило кількість холодних стартів приблизно на 27% та кількість необроблених запитів приблизно на 14%. Встановлено обмеження підходу, яке полягає у потребі великого обсягу історичних даних для навчання, евристичних порогових

значеннях масштабування та необхідності перенавчання моделі при зміні характеру навантаження.

У другому експерименті досліджено прогнозне автомасштабування на основі аналітичної моделі з напівмарковськими процесами, що безпосередньо базується на теоретичних результатах Розділу 3. Розроблений фреймворк протестовано на 1025132 записах. Обґрунтовано вибір мінімальної конфігурації пам'яті Lambda (128 MB) на основі аналізу залежності вартості від конфігурації ресурсів: при параметрі масштабування $\alpha \leq 1$ (що відповідає змішаному I/O та CPU навантаженню) мінімальна конфігурація є оптимальною за вартістю. Результати тестування показали прискорення обробки даних на 25,8%, зростання пропускної здатності на 21,3% та зменшення холодних стартів до 3% порівняно з класичним реактивним масштабуванням.

У третьому експерименті проведено імітаційне моделювання СМО методом Монте-Карло на реальних даних (578 записів, 19 інтервалів, 4 конфігурації по 100 прогонів кожна). Оцінка параметрів суміші за допомогою ЕМ-алгоритму виявила нестаціонарність вхідного процесу (кількість компонентів суміші змінюється від 1 до 4), а тести Колмогорова–Смірнова підтвердили статистичну відмінність розподілів у 44,4% пар послідовних інтервалів. Оптимальний протокол вибору серверу забезпечує статистично значуще зменшення частки відхилених заявок на 1,51% ($p = 0{,}0104$), що підтверджує аналітичний результат $L_q^{\text{opt}} < L_q$ з підпункту 3.3.4. Модель суміші потоків (3.2) демонструє менший Error Rate (0,01896) порівняно з MMPP-наближенням (0,01915), що узгоджується з теоретичною нерівністю $\text{Var}\big(N(t)\big) \leq \text{Var}\left(\widetilde{N}(t)\right)$ з підпункту 3.1.3. Найкращі показники забезпечує конфігурація System 2 + Optimal (Error Rate = 0,01881 ± 0,00118).

Запропоновано алгоритм налаштування параметрів СМО (Algorithm 3), який є деталізацією Кроку 8 алгоритму параметричної оцінки з підпункту 3.3.5. Алгоритм використовує тест Колмогорова–Смірнова для прийняття

рішення про необхідність зміни конфігурації системи, що забезпечує адаптивну реакцію на зміни структури навантаження. Сформульовано багатокритеріальну задачу оптимізації, що поєднує мінімізацію вартості обслуговування $T_{cost}(S, R)$, частку відмов $\theta(T)$ та час простою D_T.

Порівняння підходів, які досліджувалися, показало, що аналітична модель на основі напівмарковських процесів перевищує ML-підхід (DeepAR) за основними показниками продуктивності та забезпечує аналітичні гарантії якості обслуговування.

Основні результати розділу опубліковані у працях [71, 103, 111, 129, 153, 169].

ВИСНОВКИ

У результаті виконання дисертаційного дослідження розв’язано актуальну науково-прикладну задачу оптимізації безсерверних обчислень у хмарних середовищах шляхом побудови інформаційної технології як неоднорідної системи масового обслуговування та розробки фреймворку для прогнозного автоматичного масштабування обчислювальних ресурсів. У роботі отримано наступні наукові та практичні результати.

1. Здійснено аналіз сучасних підходів до реалізації, масштабування та оцінки ефективності безсерверних обчислень, визначено їх переваги та обмеження. Досліджено архітектурні шаблони подієво-орієнтованих систем, способи комунікації між компонентами та критерії вибору архітектурних рішень. Встановлено, що існуючі аналітичні моделі безсерверних платформ використовують стаціонарні моделі черг з однорідним вхідним потоком, що не враховує гетерогенність джерел подій та нестаціонарність навантаження, характерні для реальних безсерверних систем. Це обґрунтувало необхідність розробки нових моделей на основі неоднорідних систем масового обслуговування зі змішаними режимами роботи та дозволило сформулювати задачі дисертаційного дослідження.

2. Вперше побудовано та досліджено математичну модель безсерверної обчислювальної системи на прикладі AWS Lambda як неоднорідної системи масового обслуговування $H_k(t)/M/\infty$ зі змішаними режимами роботи, що, на відміну від відомих моделей, враховує гетерогенність джерел подій та стохастичну природу процесів надходження та обробки завдань. Запропонована модель дозволила описати вхідний процес як суміш незалежних потоків від різних джерел (API Gateway, S3, SQS), процес обробки з експоненціальним часом, змінними кількістю серверів та інтенсивностями серверів, два режими роботи черги (необмежена для асинхронних та обмежена для синхронних викликів), а також вартісні характеристики серверів.

3. Розроблено модель суміші потоків завдань у безсерверних обчисленнях, що формується як поєднання k незалежних неоднорідних вхідних зважених пуассонівських процесів $N(t) = \sum_{i=1}^{k} w_i\,(t)N_i(t)$ з відповідними вагами. Досліджено відмінності запропонованої моделі від класичного Markov-Modulated Poisson Process (MMPP), а саме показано, що дисперсія сумарного процесу задовольняє нерівність $\mathrm{Var}\big(N(t)\big) \le \mathrm{Var}\big(\text{₦}(t)\big)$, що має суттєве значення для точності оцінок довжини черги та ймовірності перевищення лімітів одночасного виконання. Перевага запропонованої моделі підтверджена імітаційним моделюванням методом Монте-Карло, де модель суміші демонструє менший Error Rate (0,01896) порівняно з MMPP-наближенням (0,01915).

4. Вперше для побудованої моделі Hk(t)/M/∞ доведено граничні еволюції для процесу довжини черги у схемі усереднення та схемі дифузійної апроксимації (Теорема 3.2.3) з використанням апарату напівмарковських випадкових еволюцій. Встановлено необхідну та достатню умову обмеженості черги (Лема 3.2.1). На відміну від відомих результатів для стаціонарних СМО, отримані граничні теореми враховують нестаціонарність та неоднорідність вхідного процесу, що характерні для безсерверних систем. Це дозволило отримати нормальне наближення для розподілу довжини черги та формулу для оцінки частки відхилених запитів (throttling rate), яка має пряме практичне застосування для визначення відповідності вимогам SLA.

5. Удосконалено алгоритм параметричної оцінки та оптимізації конфігурації безсерверної системи, доповнений алгоритмом динамічного налаштування параметрів СМО на основі тесту Колмогорова–Смірнова. На відміну від існуючих підходів на основі машинного навчання, які потребують великих обсягів історичних даних та перенавчання при зміні характеру навантаження, запропоновані алгоритми використовують аналітичні оцінки теорії масового обслуговування, дозволяючи отримати явні формули для оптимальних параметрів provisioned concurrency.

6. Розроблено метод оцінки параметрів суміші вхідного процесу $\left(w_i(t), \lambda_i(t)\right)$ на основі EM-алгоритму з використанням метрик AWS CloudWatch та SQS. Досліджено два протоколи розподілу завдань між серверами з різною потужністю – випадковий та оптимальний (вибір найпотужнішого вільного сервера). На відміну від існуючих евристичних підходів до балансування навантаження, для *M/M/2* системи аналітично доведено, що оптимальний протокол забезпечує меншу середню довжину черги ($L_q^{\text{opt}} < L_q$) та менший час очікування ($W_q^{\text{opt}} < W_q$). Сформульовано задачу оптимального розподілу вхідних потоків між групами серверів за критерієм рівномірного завантаження.

7. Розроблено архітектуру фреймворку для розподіленої обробки даних з використанням безсерверної архітектури, що, на відміну від існуючих рішень з реактивним масштабуванням, включає модуль прогнозного автоматичного масштабування. Фреймворк використовує AWS SQS як чергу повідомлень, AWS Lambda для подієво-орієнтованої обробки, DynamoDB для збереження стану та AppSync для моніторингу в реальному часі. Порівняння підходів на реальних даних показало, що аналітична модель перевищує ML-підхід за ключовими показниками та забезпечує гарантії SLA.

8. Реалізовано інформаційну технологію для розподіленої обробки даних з використанням безсерверних обчислень та проактивним автоматичним масштабуванням обчислювальних ресурсів. Прогнозне автомасштабування на основі аналітичної моделі забезпечило прискорення обробки даних на 25,8%, зростання пропускної здатності на 21,3% та зменшення холодних стартів до 3% порівняно з класичним реактивним масштабуванням (1 025 132 записи).

9. Проведено експериментальну апробацію розроблених моделей та інформаційної технології у хмарному середовищі AWS на основі трьох серій експериментів. Імітаційне моделювання методом Монте-Карло (4 конфігурації по 100 прогонів на реальних даних) підтвердило теоретичні результати: нестаціонарність вхідного процесу (44,4% пар інтервалів мають

статистично різні розподіли за тестом Колмогорова–Смірнова), перевагу оптимального протоколу вибору серверу ($p = 0{,}0104$) та перевагу моделі суміші потоків над MMPP-наближенням.

10. Результати дисертаційної роботи впроваджені у роботу Finker Finance B.V. та ФОП Вербицької С.І. та використовуються в освітньому процесі відділу комп'ютерних технологій Навчально-наукового інституту фізико-технічних та комп'ютерних наук Чернівецького національного університету імені Юрія Федьковича.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Serverless Computing: Current Trends and Open Problems / I. Baldini et al. *ArXiv preprint arXiv:1706.03178*. 2017. URL: https://doi.org/10.48550/arXiv.1706.03178.

2. The Rise of Serverless Computing / P. Castro et al. *Communications of the ACM*. 2019. Vol. 62, № 12. P. 44–54. URL: https://doi.org/10.1145/3368454.

3. Cloud Programming Simplified: A Berkeley View on Serverless Computing / E. Jonas et al. *ArXiv preprint arXiv:* abs/1902.03383. 2019. URL: https://doi.org/10.48550/arXiv.1902.03383.

4. Serverless Computing: One Step Forward, Two Steps Back / J. M. Hellerstein et al. *arXiv preprint arXiv:*1812.03651v1. 2018. URL: https://doi.org/10.48550/arXiv.1812.03651.

5. Mampage A. Autonomous Resource Management for Serverless Computing : Doctor of Philosophy thesis / School of Computing and Information Systems, The University of Melbourne. Melbourne, 2023. 248 p. URL: https://clouds.cis.unimelb.edu.au/students/AnupamaPhD_Thesis2023.pdf (дата звернення: 02.02.2026).

6. Mukherjee A., Buyya R. Federated Learning Architectures: A Performance Evaluation With Crop Yield Prediction Application. *Software: Practice and Experience*. 2025. Vol. 55, Issue 7. P. 1165–1184. URL: https://doi.org/10.1002/spe.3416.

7. Atoll: A Scalable Low-Latency Serverless Platform / A. Singhvi et al. *SoCC'21* : Proceedings of the ACM Symposium on Cloud Computing, Seattle WA USA, Nov. 1–4, 2021 / Association for Computing Machinery. New York, NY, USA : ACM, 2021. P. 138–152. URL: https://doi.org/10.1145/3472883.3486981.

8. Spock: Exploiting serverless functions for SLO and cost aware resource procurement in public cloud / J. R. Gunasekaran et al. *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)* : Proccedings or the Int. Conf., Milan, Italy, July 8–13, 2019 / IEEE. Piscataway, NJ, USA : IEEE, 2019. P. 199–208. URL: https://doi.org/10.1109/CLOUD.2019.00043.

9. Nestorov A. M. Optimizing serverless architectures for data-intensive analytics workloads : Doctor of Philosophy thesis in the subject of Computer Architecture / Universitat Politècnica de Catalunya, Departament d'Arquitectura de Computadors. 2024. 129 p. URL: https://doi.org/10.5821/dissertation-2117-409989.

10. Cirrus: A serverless framework for end-to-end ML workflows / J. Carreira et al. *SoCC'19* : Proceedings of the ACM Symposium on Cloud Computing, Santa Cruz, CA, USA, Nov. 20–23, 2019 / Association for Computing Machinery. New York, NY, USA : ACM, 2019. P. 13–24. URL: https://doi.org/10.1145/3357223.3362711.

11. Predoaia I., García-López P. A cloud-agnostic serverless architecture for distributed machine learning. *BDCAT'24* : Proceedings of the 2024 IEEE/ACM International Conference on Big Data Computing, Applications and Technologies, Sharjah, United Arab Emirates, Dec. 16–19, 2024 / IEEE. Piscataway, NJ, USA : IEEE, 2024. P. 131–140.
URL: https://doi.org/10.1109/BDCAT63179.2024.00032.

12. Wang H., Niu D., Li B. Distributed machine learning with a serverless architecture. *IEEE INFOCOM 2019 – IEEE Conference on Computer Communications :* Proccedings or the Int. Conf., Paris, France, April 29–May 2, 2019 / Piscataway, NJ, USA : IEEE, 2019. P. 1288–1296. URL: https://doi.org/10.1109/INFOCOM.2019.8737391.

13. Stellaris: Staleness-aware distributed reinforcement learning with serverless computing / H. Yu et al. *SC24* : Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Atlanta, GA, USA, November 17–22, 2024 / IEEE. Piscataway, NJ, USA : IEEE. P. 1–17. URL: https://doi.org/10.1109/SC41406.2024.00045.

14. Nguyen T. Holistic cold-start management in serverless computing cloud with deep learning for time series. *Future Generation Computer Systems.* 2024. Vol. 153. P. 312–325. URL: https://doi.org/10.1016/j.future.2023.12.011.

15. Cold Start Latency in Serverless Computing: A Systematic Review, Taxonomy, and Future Directions / M. Golec et al. *ACM Computing Surveys*. 2024. Vol. 57, Issue 3, № 65. P. 1–36. URL: https://doi.org/10.1145/3700875.

16. Schuler L., Jamil S., Kuhl N. AI-based resource allocation: Reinforcement learning for adaptive auto-scaling in serverless environments. *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid) :* Proccedings or the Int. Conf., Melbourne, Australia, 2021 / IEEE. Piscataway, NJ, USA : IEEE, 2021. P. 804–811. URL: https://doi.org/10.1109/CCGrid51090.2021.00098.

17. Koroliuk V. S., Koroliuk V. V., Limnios N. Queuing Systems with Semi-Markov Flow in Average and Diffusion Approximation Schemes. *Methodology and Computing in Applied Probability.* 2009. Vol. 11. P. 201–209. URL: https://doi.org/10.1007/s11009-008-9081-7.

18. Malyk I. V. Compensating Operator and Weak Convergence of Semi-Markov Process to the Diffusion Process without Balance Condition. *Journal of Applied Mathematics*. 2015. Art. 563060. 7 p. URL: https://doi.org/10.1155/2015/563060.

19. Bezverbnyi I., Shyshkina M. The Use of Serverless Technologies to Support Data Processing within the Open Learning and Research Systems. In *Proceedings of the 1st Symposium on Advances in Educational Technology. /* SCITEPRESS – Science and Technology Publications. Setubal, Portugal : SCITEPRESS, 2022. Vol. 2. P. 489–494. URL: https://doi.org/10.5220/0010933200003364.

20. Shafiei H., Khonsari A., Mousavi P. Serverless Computing: A Survey of Opportunities, Challenges, and Applications. *ACM Computing Surveys*. 2022. Vol. 54, Issue 11s. Art. 239. P. 1–32. URL: https://doi.org/10.1145/3510611.

21. Status of Serverless Computing and Function-as-a-Service (FaaS) in Industry and Research / G. C. Fox et al. *arXiv preprint arXiv:1708.08028v1*. 2017. URL: https://doi.org/10.48550/arXiv.1708.08028.

22. What Serverless Computing Is and Should Become: The Next Phase of Cloud Computing / J. Schleier-Smith et al. *Communications of the ACM. 2021*. Vol. 64, Issue 5. P. 76–84. URL: https://doi.org/10.1145/3406011.

23. Adzic G., Chatley R. Serverless computing: Economic and architectural impact. *ESEC/FSE 2017* : Proceedings of the 2017 11th Joint Meeting on Foundations of Software, Paderborn, Germany, Sept. 4–8, 2017 / Association for Computing Machinery. New York, NY, USA : ACM, 2017. P. 884–889. URL: https://doi.org/10.1145/3106237.3117767.

24. Serverless Computing at the Edge for AIoT Applications / H. Mora et al. *ICAIoT :* Proceedings of the 2022 International Conference on Artificial Intelligence of Things (ICAIoT), Istanbul, Turkey, Dec. 29–30, 2022 / IEEE. Piscataway, NJ, USA : IEEE, 2022. P. 1–6. URL: https://doi.org/10.1109/ICAIoT57170.2022.10121879.

25. SoK: Function-as-a-Service: From an Application Developer's Perspective / A. Raza A et al. *Journal of Systems Research.* 2021. Vol. 1, Issue 1. 20 p. URL: https://doi.org/10.5070/SR31154815.

26. SFS: smart OS scheduling for serverless functions / Y. Fu et al. *SC'22* : Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, Dallas, Texas, Nov. 13–18, 2022 / IEEE. Piscataway, NJ, USA : IEEE, 2022. Article 42. P. 1–16.

27. Miller R. Amazon Launches Lambda, An Event-Driven Compute Service. *TechCrunch :* website. URL: https://techcrunch.com/2014/11/13/amazon-launches-lambda-an-event-driven-compute-service/ (дата звернення: 02.02.2026).

28. Roberts M. Serverless Architectures. *MartinFowler.com* : website. URL: https://martinfowler.com/articles/serverless.html (дата звернення: 02.02.2026).

29. Lee H., Satyam K., Fox G. Evaluation of production serverless computing environments. *2018 IEEE 11th International Conference on Cloud Computing (CLOUD) :* Proccedings or the Int. Conf., San Francisco, CA, USA, July 2–7, 2018 / IEEE. Piscataway, NJ, USA : IEEE, 2018. P. 442–450. URL: https://doi.org/10.1109/CLOUD.2018.00062.

30. Scheuner J., Leitner P. Function-as-a-service performance evaluation: A multivocal literature review. *Journal of Systems and Software*. 2020. Vol. 170. Art. 110708. 20 p. URL: https://doi.org/10.1016/j.jss.2020.110708.

31. Peeking Behind the Curtains of Serverless Platforms / L. Wang et al. *USENIX ATC'18 :* Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18), Boston, MA, USA, July 11–13, 2018 /USENIX Association. Berkeley, CA, USA : USENIX Association, 2018. P. 133–145. URL: https://dl.acm.org/doi/10.5555/3277355.3277369.

32. Jawaddi S. N. A., Ismail A. Autoscaling in Serverless Computing : Taxonomy and OpenChallenges. *Research Square*. 12.05.2023. URL: https://www.researchsquare.com/article/rs-2897886/v1 (дата звернення: 03.03.2026).

33. Кириченко О. Л., Кириченко О. О. Використання AWS APPSYNC для комунікації вебдодатків у реальному часі. *Вчені записки Таврійського національного університету імені В.І. Вернадського.* Серія : Технічні науки. 2024. 35(74), №4. С. 105–110. URL: https://doi.org/10.32782/2663-5941/2024.4/16.

34. Kyrychenko O., Kyrychenko O. Real-time communication tools for web applications in a cloud environment. *The 13 th International Conference on Electronics, Communications and Computing's (IC ECCO)* : Proccedings of the Intern. Conf., Chisinau, Moldova, October 17–18, 2024 / IEEE. Piscataway, NJ, USA : IEEE, 2024. P. 127–128.
URL: https://repository.utm.md/bitstream/handle/5014/28779/Int-Conf-ECCO-2024-Abstract-Book-p126-127.pdf?sequence=1&isAllowed=y

35. Patterns for Serverless Functions (Function-as-a-Service): A Multivocal Literature Review / D. Taibi et al. *CLOSER 2020* : Proceedings of the 10th International Conference on Cloud Computing and Services Science *(CLOSER 2020)*, Prague, Czech Republic (online), May 7–9, 2020 / SCITEPRESS – Science and Technology Publications. Setubal, Portugal : SCITEPRESS, 2020. P. 181–192. URL: https://www.scitepress.org/Link.aspx?doi=10.5220/0009578501810192.

36. Faasten your decisions: A classification framework and technology review of function-as-a-service platforms / V. Yussupovet al. *Journal of Systems and Software*. 2021. Vol. 175. Art. 110906. 24 p. URL: https://doi.org/10.1016/j.jss.2021.110906.

37. The state of serverless applications: Collection, characterization, and community consensus / Eismann S. et al. *IEEE Transactions on Software Engineering.* 2021. Vol. 48, № 10. P. 4152-4166. URL: https://scispace.com/pdf/the-state-of-serverless-applications-collection-1o28zt7c.pdf. (дата звернення: 21.01.2026)

38. Kim J., Lee K. FunctionBench: A suite of workloads for serverless cloud function service. *2019 IEEE 12th International Conference on Cloud Computing (CLOUD) :* Proccedings of the Intern. Conf., Milan, Italy, July 8–13, 2019 / IEEE. Piscataway, NJ, USA : IEEE, 2019. P. 502–504. URL: https://doi.org/10.1109/CLOUD.2019.00091.

39. Characterizing serverless platforms with ServerlessBench / T. Yu et al. *SoCC'20* : Proceedings of the 11th ACM Symposium on Cloud Computing, Virtual Event, USA, Oct. 19–21, 2020 / Association for Computing Machinery. New York, NY, USA : ACM, 2020. P. 30–44. URL: https://doi.org/10.1145/3419111.3421280.

40. SEBS: A serverless benchmark suite for function-as-a-service computing / M. Copik et al. *Middleware'21* : Proceedings of the 22nd International Middleware Conference, Québec city, Canada, Dec. 6–10, 2021 / Association for Computing Machinery. New York, NY, USA : ACM, 2021. P. 64–78. URL: https://doi.org/10.1145/3464298.3476133.

41. Gallardo S. R. Serverless Strategies and Tools in the Cloud Computing Continuum : Doctor of Philosophy thesis in the subject of Computer Science / Universitat Politècnica de València. 2023. 168 p. URL: https://dialnet.unirioja.es/servlet/tesis?codigo=328486 (дата звернення: 02.02.2026).

42. Triendl S., Ristov S. Peeking Behind the Serverless Implementations and Deployments of the Montage Workflow. *ICPE'24 Companion* : Companion of the 15th ACM/SPEC International Conference on Performance Engineering, London, United Kingdom, May 7–11, 2024 / Association for Computing Machinery. New York, NY, USA : ACM, 2024. P. 196–203. URL: https://doi.org/10.1145/3629527.3651420.

43. Thatikonda K. Comprehensive Cloud Migration Strategies: Optimizing for Reliability, Security, and Performance. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*. 2025. Vol. 11, № 2. P. 72–82. URL: https://doi.org/10.32628/cseit251112388

44. Auto-Scaling Techniques in Cloud Computing: Issues and Research Directions / S. Alharthi et al. *Sensors*. 2024. Vol. 24, Issue 17. Art. 5551. 21 p. URL: https://doi.org/10.3390/s24175551.

45. Poka V. Optimizing Serverless Computing: Enhancing Performance and Efficiency in Modern Cloud Architecture. *International Journal of Scientific and Applied Technology.* 2025. Vol. 16, Issue 2. 10 p. URL: https://doi.org/10.71097/IJSAT.v16.i2.3054.

46. Кириченко О. Л., Кириченко О. О. Покращення швидкодії безсерверних додатків за допомогою кешування. *Problems of Science and Technology: the Search for Innovative Solutions :* Proceedings of the XXIII International scientific and practical conference, Munich, Germany, 15–17 May, 2024 / International Scientific Unity. Munich, Germany : ISU, 2024. P. 65–67. URL: https://isu-conference.com/wp-content/uploads/2024/05/Problems_of_science_and_technology_the_search_for_innovative_solutions_May_15_17_2024_Munich_Germany.pdf (дата звернення: 20.01.2026).

47. Build A Serverless Real Time Data Processing Application on AWS / Dr. Masrath Begum et al. *International Journal of Research Publication and Reviews.* 2023. Vol. 4, № 6. P. 3592–3596.

48. Costless: Optimizing cost of serverless computing through function fusion and placement / T. Elgamal et al. *2018 IEEE/ACM Symposium on Edge Computing (SEC) :* Proccedings of the Intern. Conf., Seattle, WA, USA, Oct. 25–27, 2018/ IEEE. Piscataway, NJ, USA : IEEE, 2018. P. 300–312. URL: https://doi.org/10.1109/SEC.2018.00029.

49. Configuring serverless functions using statistical learning / A. Raza et al. *IEEE Transactions on Network and Service Management*. 2023. Vol. 20, № 2. P. 1065-1077. URL: https://doi.org/10.1109/TNSM.2023.3254437.

50. Gupta A. A Predictive Framework for Managing DevOps Practices Using Machine Learning Models. *IRE Journals*. 2021. Vol. 4, № 9. P. 362–373. URL: https://www.researchgate.net/publication/395340605_A_Predictive_Framework_for_Managing_DevOps_Practices_Using_Machine_Learning_Models (дата звернення: 19.01.2026).

51. A Serverless Real-Time Data Analytics Platform for Edge Computing / S. Nastic et al. *IEEE Internet Computing.* 2017. Vol. 21, № 4. P. 64–71. URL: https://doi.org/10.1109/MIC.2017.2911430.

52. Kaur N., Mittal A. Fog Computing Serverless Architecture for Real Time Unpredictable Traffic. *IOP Conference Series. Materials Science and Engineering* : Proccedings of the Intern. Conf., Rajpura, India, Oct., 24th, 2020 / IOP Publishing. Bristol, UK : IOP Publishing, 2021. Vol. 1022, Art. 012026. 8 p. URL: https://doi.org/10.1088/1757-899X/1022/1/012026.

53. Ghorbian M., Ghobaei-Arani M. A survey on the cold start latency approaches in serverless computing: an optimization-based perspective. *Computing.* 2024. Vol. 106, Issue 11. P. 3755–3809. URL: https://doi.org/10.1007/s00607-024-01335-5.

54. Chadha M., Jindal A., Gerndt M. Architecture-Specific Performance Optimization of Compute-Intensive FaaS Functions. *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)* : Proccedings of the Intern. Conf.,

Chicago, IL, USA, Sept. 5–10, 2021 / IEEE. Piscataway, NJ, USA : IEEE, 2021. P. 478–483. URL: https://doi.org/10.1109/CLOUD53861.2021.00062.

55. Thumala S. Building Highly Resilient Architectures in the Cloud. *Nanotechnology Perceptions*. 2020. Vol. 16. P. 264–284.

56. Auto-scaling mechanisms in serverless computing: A comprehensive review / M. Tari et al. *Computer Science Review*. 2024. Vol. 53. Art. 100650. URL: https://doi.org/10.1016/j.cosrev.2024.100650.

57. Cordingly R., Shu W., Lloyd W. J. Predicting performance and cost of serverless computing functions with SAAF. *2020 IEEE International Conference on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)* : Proccedings of the Intern. Conf., Calgary, AB, Canada, Aug. 17–22, 2020 / IEEE. Piscataway, NJ, USA : IEEE, 2020. P. 640–649. URL: https://doi.org/10.1109/DASC-PICom-CBDCom-CyberSciTech49142.2020.00111.

58. Improving Application Migration to Serverless Computing Platforms: Latency Mitigation with Keep-Alive Workloads / W. Lloyd et al. *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)* : Proccedings of the Intern. Conf., Zurich, Switzerland, Dec. 17–20, 2018 / IEEE. Piscataway, NJ, USA : IEEE, 2018. P. 195–200, URL: https://doi.org/10.1109/UCC-Companion.2018.00056.

59. Mohammed C. M., Zeebaree S. R. Sufficient Comparison Among Cloud Computing Services: IaaS, PaaS, and SaaS: A Review. *International journal of science and business.* 2021. Vol. 5, Issue 2. P. 17–30. URL: https://doi.org/10.5281/zenodo.4450129.

60. An analysis of security issues for cloud computing / K. Hashizume et al. *Journal Internet Services and Applications.* 2013. Vol. 4, №5. URL: https://doi.org/10.1186/1869-0238-4-5.

61. Opara-Martins J., Sahandi R., Tian F. Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective. *Journal of Cloud Computing*. 2016. Vol. 5, №4. 18 p. URL: https://doi.org/10.1186/s13677-016-0054-z.

62. Ebrahimi A., Ghobaei-Arani M., Saboohi H. Cold start latency mitigation mechanisms in serverless computing: Taxonomy, review, and future directions. *Journal of Systems Architecture*. 2024. Vol. 151. Art. 103115. URL: https://doi.org/10.1016/j.sysarc.2024.103115.

63. Karamzadeh A., Shameli-Sendi A. Reducing cold start delay in serverless computing using lightweight virtual machines. *Journal of Network and Computer Applications*. 2024. Vol. 232. Art. 104030. URL: https://doi.org/10.1016/j.jnca.2024.104030.

64. Sbarski P. Serverless architectures on AWS. Shelter Island : Manning Publications, 2017. 376 p.

65. Антонюк Д. В., Кириченко О. О. Пакетна обробка даних на безсерверній архітектурі. *Проблеми інформатики та комп'ютерної техніки (ПІКТ–2023)* : праці XII праці Міжнар. наук.-практ. конф., м. Чернівці, 10–12 листопада 2023 / Чернівці : Черн. нац. ун-т, 2023. С. 106–109.

66. Lefebvre M. Reducing the Size of a Waiting Line Optimally. *WSEAS Transactions on Systems and Control*. 2023, Vol. 18. P. 342-345. URL: https://doi.org/10.37394/23203.2023.18.35.

67. Enhanced Runtime-Adaptable Routing for Serverless Functions Based on Performance and Cost Tradeoffs in Hybrid Cloud Settings / G. Fatouros et al. 2023 *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)* : Proccedings of the Intern. Conf., Naples, Italy, Dec. 4–6, 2023 / IEEE. Piscataway, NJ, USA : IEEE, 2023. P. 177–184 URL: https://doi.org/10.1109/CloudCom59040.2023.00038.

68. Tantalaki N., Souravlas S., Roumeliotis M. A review on Big Data real-time stream processing and its scheduling techniques. *International Journal of*

Parallel, Emergent and Distributed Systems. 2020. Vol. 35, № 5. P. 571–601. URL: https://doi.org/10.1080/17445760.2019.1585848.

69. Move Fast and Meet Deadlines: Fine-grained Real-time Stream Processing with Cameo / L. Xu et al. *arXiv preprint arXiv: 2010.03035v1*. 2020. URL: https://arxiv.org/abs/2010.03035.

70. Santhi K., Saravanan R. Performance analysis of cloud computing bulk service using queueing models. *International Journal of Applied Engineering Research*. 2017. Vol. 12, № 17. P. 6487–6492.

71. Кириченко О.О., Кириченко О. Л., Остапов С.Е. Аналіз продуктивності AWS SQS в умовах високих навантажень. *Проблеми інформатики та комп'ютерної техніки (ПІКТ–2024)* : праці XIII Міжнар. наук.-практ. конф., м. Чернівці, 01–03 листопада 2024 р. Чернівці : Черн. нац. ун-т, 2024. С. 42–44.

72. Kyrychenko Oleksandr O., Ostapov Sergey E., Kyrychenko Oksana L. Optimization of SQS Configurations for Efficient Batch Data Processing. *WSEAS Transactions on Systems.* 2025. Vol. 24. P. 36–43. URL: https://doi.org/10.37394/23202.2025.24.4.

73. Modeling and Estimation of Traffic Intensity in *M/M/1* Queueing System with Balking: Classical and Bayesian Approaches / B. Kushvaha et al. *AppliedMath.* 2025. Vol. 5, Issue 1. Art. 19. URL: https://doi.org/10.3390/appliedmath5010019.

74. A hybrid queuing model for Virtual Machine placement in cloud data center / S. K. Addya et al. *2015 IEEE International Conference on Advanced Networks and Telecommuncations Systems (ANTS)* : Proccedings of the Intern. Conf., Kolkata, India, Dec. 15–18, 2015 / IEEE. Piscataway, NJ, USA : IEEE, 2015. P. 1–3. URL: https://doi.org/10.1109/ANTS.2015.7413642.

75. FAAStloop: Optimizing Loop-Based Applications for Serverless Computing / S. Mohanty et al. *SoCC'24* : Proceedings of the 2024 ACM Symposium on Cloud Computing, WA, USA, Nov. 20–22, 2024 / Association for

Computing Machinery. New York, NY, USA : ACM, 2024. P. 943–960. URL: https://doi.org/10.1145/3698038.3698560.

76. Skedulix: Hybrid cloud scheduling for cost-efficient execution of serverless applications A. Das et al. *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)* : Proceedings of the Intern. Conf., Beijing, China (online), Oct. 19–24, 2020 / IEEE. Piscataway, NJ, USA : IEEE, 2020. P. 609–618. URL: https://doi.org/10.1109/CLOUD49709.2020.00090.

77. ENSURE: Efficient scheduling and autonomous resource management in serverless environments / A. Suresh et al. *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)* : Proceedings of the Intern. Conf., Washington, DC, USA (online), Aug. 17–21, 2020 / IEEE. Piscataway, NJ, USA : IEEE, 2020. P. 1–10. URL: https://doi.org/10.1109/ACSOS49614.2020.00020.

78. Ghahramani M. H., Zhou M., Hon C. T. Toward cloud computing QoS architecture: analysis of cloud systems and cloud services. *IEEE/CAA Journal of Automatica Sinica*. 2017. Vol. 4, № 1. P. 6–18. URL: https://doi.org/10.1109/JAS.2017.7510313.

79. CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms / R. N. Calheiros et al. *Software: Practice and Experience*. 2011. Vol. 41, Issue 1. P. 23–50. URL: https://doi.org/10.1002/spe.995.

80. Serverless is More: From PaaS to Present Cloud Computing / E. van Eyk et al. *IEEE Internet Computing*. 2018. Vol. 22, № 5. P. 8–17. URL: https://doi.org/10.1109/MIC.2018.053681358.

81. Predicting the costs of serverless workflows / S. Eismann et al. *ICPE '20* : Proceedings of the ACM/SPEC International Conference on Performance Engineering, Edmonton, AB, Canada (online), Apr. 20–24, 2020 / Association for Computing Machinery. New York, NY, USA : ACM, 2020. P. 265–276. URL: https://doi.org/10.1145/3358960.3379133.

82. Кириченко О., Кириченко О. Кешування даних у додатках з використанням безсерверної архітектури. *Information Technology : Computer Science Software Engineering and Cyber Security*. 2024. № 2. С. 42–49. URL: https://doi.org/10.32782/IT/2024-2-6.

83. O'Riordan M., Wiggers S.-J. Using Serverless WebSockets to Enable Real-Time Messaging. *InfoQ* : website. URL: https:// www.infoq.com/articles/serverless-websockets-realtime-messaging/ (дата звернення: 23.10.2025).

84. Serverless in the wild: characterizing and optimizing the serverless workload at a large cloud provider / M. Shahrad et al. *USENIX ATC'20* : Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference, Boston, MA, USA (online), July 15–17, 2020 / USENIX Association. Berkeley, CA, USA : USENIX Association, 2020. Article 14. P. 205–218.

85. An optimal delay aware task assignment scheme for wireless SDN networked edge cloudlets / S.S. Chalapathi G.et al. *Future Generation Computer Systems*. 2020. Vol. 102. P. 862–875. URL: https://doi.org/10.1016/j.future.2019.09.003.

86. Spohn M. A. Publish, subscribe, and federate! *arXiv preprint arXiv:2006.03675v1.* 2020. URL: https://doi.org/10.48550/arXiv.2006.03675.

87. Designing for Scalability: Building a Universal Serverless Messaging Architecture with Apache RocketMQ / J. Ji et al. *FSE Companion`25* : Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering**,** Tokyo, Japan, June 23–27, 2025 / Association for Computing Machinery. New York, NY, USA : ACM, 2025. P. 105–110. URL: https://doi.org/10.1145/3696630.3728536.

88. Benchmarking Message Queues / R. Maharjan et al. *Telecom.* 2023. Vol. 4, Issue 2. P. 298–312. URL: https://doi.org/10.3390/telecom4020018.

89. Neelan A. The Evolving Role of API Gateways in Scalable Microservices Architecture. *International Journal of Emerging Trends in*

Computer Science and Information Technology. 2025. Vol. 6, № 4. P. 4–15. URL: https://doi.org/10.63282/3050-9246.IJETCSIT-V6I4P102.

90. Velepucha V., Flores P. A Survey on Microservices Architecture: Principles, Patterns and Migration Challenges. *IEEE Access*. 2023. Vol. 11. P. 88339-88358. URL: https://doi.org/10.1109/ACCESS.2023.3305687.

91. Patsch R., Göschka K. M. Make Applications FaaS-ready: Challenges and Guidelines. *ITCC'24* : Proceeding of the 2024 6th International Conference on Information Technology and Computer Communications, Singapore, Singapore, Aug. 14–16, 2024 / Association for Computing Machinery. New York, NY, USA : ACM, 2025. P. 57–64. URL: https://doi.org/10.1145/3704391.3704400.

92. Patsch R., Göschka K. M. Performance-Centric Communication in Serverless Functions With TCP Hole Punching. *ICCTA'24* : Proceedings of the 2024 10th International Conference on Computer Technology Applications, Vienna, Austria, May 15–17, 2024 / Association for Computing Machinery. New York, NY, USA : ACM, 2024. P. 109–115. URL: https://doi.org/10.1145/3674558.3674573.

93. Liu Q., Sun X. Research of Web Real-Time Communication Based on Web Socket. *International Journal of Communications, Network and System Sciences*. 2012. Vol. 5, № 12. P. 797–801. URL: https://doi.org/10.4236/ijcns.2012.512083.

94. Amazon API Gateway. *AWS* : website. URL: https://aws.amazon.com/api-gateway/ (дата звернення: 23.01.2026).

95. Amazon DynamoDB. *AWS* : website. URL: https://aws.amazon.com/dynamodb/ (дата звернення: 23.01.2026).

96. Configuring Real-Time Event Processing of Api Gateway With Aws and Websocket Api's / K. S. Sarat Dyuthi et al. *Journal of Informatics Education and Research*. 2024. Vol. 4, № 3. P. 3324–3337. URL: https://doi.org/10.52783/jier.v4i3.1711.

97. Gorantla V. K. C. A Hybrid WebSocket-REST Approach for Scalable Real-Time API Design`. *International Journal of Emerging Trends in Computer*

Science and Information Technology. 2021. Vol. 2, № 3. P. 60–69. URL: https://doi.org/10.63282/3050-9246.IJETCSIT-V2I3P107.

98. What is AWS AppSync? *AWS* : website. URL: https://docs.aws.amazon.com/appsync/latest/devguide/what-is-appsync.html (дата звернення: 22.01.2026).

99. Lima Ed. New features that will enhance your Real-Time experience on AWS AppSync. *AWS* : website. URL: https://aws.amazon.com/blogs/mobile/appsync-realtime/ (дата звернення: 22.01.2026).

100. Manchana R. K. Operationalizing Batch Workloads in the Cloud with Case Studies. *International Journal of Science and Research (IJSR)*. 2020. Vol. 9, Issue 7. P. 2031–2041. URL: https://doi.org/10.21275/SR24820052154.

101. Proactive Auto-Scaling for Service Function Chains in Cloud Computing Based on Deep Learning / M. B. Taha et al. *IEEE Access*. 2024. Vol. 12. P. 38575–38593. URL: https://doi.org/10.1109/ACCESS.2024.3375772.

102. Verma S., Bala A. Auto-scaling techniques for IoT-based cloud applications: a review. *Cluster Computing*. 2021. Vol. 24. P. 2425–2459. URL: https://doi.org/10.1007/s10586-021-03265-9.

103. Kyrychenko O., Ostapov S., Kyrychenko O. Design of a framework for serverless distributed data processing using queues. *Eastern-European Journal of Enterprise Technologies*. 2025. Vol. 4, № 9. P. 19–25. URL: https://doi.org/10.15587/1729-4061.2025.335723.

104. Nastic S. Self-Provisioning Infrastructures for the Next Generation Serverless Computing. *SN Computer Science.* 2024. Vol. 5, Art. 678. 15 p. URL: https://doi.org/10.1007/s42979-024-03022-w.

105. The Gap Between Serverless Research and Real-world Systems / Q. Liu et al. *SoCC '23* : Proceedings of the 2023 ACM Symposium on Cloud Computing, Santa Cruz, CA, USA, Oct. 30–Nov. 1, 2023 / Association for Computing Machinery. New York, NY, USA : ACM, 2023. P. 475–485. URL: https://doi.org/10.1145/3620678.3624785.

106. Amazon Simple Storage Service Documentation. *AWS* : website. URL: https://docs.aws.amazon.com/s3/ (дата звернення: 22.01.2026).

107. Amazon Simple Queue Service. *AWS* : website. URL: https://aws.amazon.com/sqs/ (дата звернення: 22.01.2026).

108. What is AWS Lambda? *AWS* : website. URL: https://docs.aws.amazon.com/lambda/latest/dg/welcome.html (дата звернення: 22.01.2026).

109. Amazon SageMaker. *AWS* : website. URL: https://aws.amazon.com/sagemaker/ (дата звернення: 22.01.2026).

110. Amazon EventBridge - event-driven integration service. *AWS* : website. URL: https://aws.amazon.com/eventbridge/ (дата звернення: 22.01.2026).

111. Kyrychenko O., Ostapov S., Kyrychenko O. L. Predictive autoscaling in AWS Serverless by means of machine learning and SQS metrics. *6th International Workshop on Intelligent Information Technologies & Systems of Information Security (IntelITSIS)*: CEUR Workshop Proceedings, Khmelnytskyi, Ukraine, 4th April, 2025 / CEUR-WS.org. Aachen, Germany : RWTH Aachen University, 2025.Vol.-3963. P. 77–87. URL: https://ceur-ws.org/Vol-3963/.

112. Evaluating Semi-Markov Processes and Other Epidemiological Time-to-Event Models by Computing Disease Sojourn Density as Partial Differential Equations / J. Worthington et al. *Medical decision making : an international journal of the Society for Medical Decision Making*. 2025. Vol. 45, № 5. P. 569–586. https://doi.org/10.1177/0272989X251333398.

113. Emergence of serverless computing in cloud: A state-of-the-art review of challenges and opportunities / A. Kumari et al. *Machine Learning with Applications*. 2026. Vol. 24. Art. 100862. URL: https://doi.org/10.1016/j.mlwa.2026.100862.

114. Rojas L., Yepes V., Garcia J. Complex Dynamics and Intelligent Control: Advances, Challenges, and Applications in Mining and Industrial Processes. *Mathematics*. 2025. Vol. 13, Issue 6, Art. 961. 43 p. URL: https://doi.org/10.3390/math13060961.

115. AWS Lambda - serverless event-driven compute. *AWS* : website. URL: https://aws.amazon.com/lambda/ (дата звернення: 21.01.2026).

116. Rise of the Planet of Serverless Computing: A Systematic Review / J. Wen et al. *ACM Transaction on Software Engineering and Methodology.* 2023. Vol. 32, Issue 5, Art. 131. P. 1–61. URL: https://doi.org/10.1145/3579643.

117. Mahmoudi N., Khazaei H. Performance Modeling of Serverless Computing Platforms. *IEEE Transactions on Cloud Computing*. 2022. Vol. 10, № 4. P. 2834–2847. URL: https://doi.org/10.1109/TCC.2020.3033373.

118. Mahmoudi N., Khazaei H. Performance Modeling of Metric-Based Serverless Computing Platforms. *IEEE Transactions on Cloud Computing*. 2023. Vol. 11, № 2. P. 1899–1910. URL: https://doi.org/10.1109/TCC.2022.3169619.

119. SCOPE: Performance testing for serverless computing / J. Wen et al. *ACM Transactions on Software Engineering and Methodology*. 2025. Vol. 34, № 8, Art. 227. P. 1–30. URL: https://doi.org/10.1145/3717609.

120. Shrestha R., Nisha B. Performance evaluation and comparison of microservices and serverless deployments in cloud. *2023 IEEE 8th International Conference on Smart Cloud (SmartCloud)* : Proceedings of the Int. Conf., Tokyo, Japan, July 7–9, 2023 / IEEE. Piscataway, NJ, USA : IEEE, 2023. P. 202–207. URL: https://doi.org/10.1109/SmartCloud58862.2023.00043.

121. High-performance serverless computing: A systematic literature review on serverless for HPC, AI, and big data / V. Besozzi et al. *IEEE Access.* 2025. Vol. 13. P. 195611–195656.
URL: https://doi.org/10.1109/ACCESS.2025.3633989.

122. RobustScaler: QoS-aware autoscaling for complex workloads / H. Qian et al. *arXiv preprint arXiv:2204.07197v1*. 2022. URL: https://arxiv.org/abs/2204.07197.

123. Mahmoudi N., Khazaei H. Temporal Performance Modelling of Serverless Computing Platforms. *WoSC'20* : Proceedings of the 2020 Sixth International Workshop on Serverless Computing, Delft, Netherlands (online),

Dec. 7–11, 2020 / Association for Computing Machinery. New York, NY, USA : ACM, 2021. P. 1–6. URL: https://doi.org/10.1145/3429880.3430092.

124. Creating event-driven architectures with Lambda. *AWS* : website. URL: https://docs.aws.amazon.com/lambda/latest/dg/concepts-event-driven-architectures.html (дата звернення: 21.01.2026).

125. Create a serverless file-processing app. *AWS* : website. URL: https://docs.aws.amazon.com/lambda/latest/dg/file-processing-app.html (дата звернення: 21.01.2026).

126. Ozkaya M. Publish/subscribe fan-out pattern in serverless architectures using SNS, SQS and Lambda. *Medium*. URL: https://medium.com/aws-lambda-serverless-developer-guide-with-hands/publish-subscribe-fan-out-pattern-in-serverless-architectures-using-sns-sqs-and-lambda-bccaa3abac9e (дата звернення: 21.01.2026).

127. How does it function? Characterizing long-term trends in production serverless workloads / A. Joosen et al. *SoCC'23* : Proceedings of the 2023 ACM Symposium on Cloud Computing, Santa Cruz, CA, USA, Oct. 30–Nov. 1, 2023 / Association for Computing Machinery. New York, NY, USA : ACM, 2023. P. 443–458. URL: https://doi.org/10.1145/3620678.3624783.

128. WISEFUSE: Workload characterization and DAG transformation for serverless workflows / A. Mahgoub et al. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*. 2022. Vol. 6, № 2. Art. 26. P. 1–28. URL: https://doi.org/10.1145/3530892.

129. Kyrychenko O. O., Ostapov S. E., Kyrychenko O. L. Optimization of SQS Configurations for Efficient Batch Data Processing. *WSEAS Transactions on Systems.* 2025. Vol. 24. P. 36–43. URL: https://doi.org/10.37394/23202.2025.24.4.

130. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems / Y. Gan et al. *ASPLOS'19 :* Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems,

Providence, RI, USA, April 13–17, 2019 / Association for Computing Machinery. New York, NY, USA : ACM, 2019. P. 3–18.
URL: https://doi.org/10.1145/3297858.3304013.

131. Characterizing microservice dependency and performance: Alibaba trace analysis / S. Luo et al. *SoCC'21* : Proceedings of the ACM Symposium on Cloud Computing, Seattle, WA, USA, Nov. 1–4, 2021 / Association for Computing Machinery. New York, NY, USA : ACM, 2021. P. 412–426. URL: https://doi.org/10.1145/3472883.3487003.

132. Mews S., Koslik J.-O., Langrock R. How to build your latent Markov model – the role of time and space. *arXiv preprint arXiv:2406.19157v4.* 2025. URL: https://arxiv.org/abs/2406.19157.

133. Uncovering the impact of bursty workloads on system performance in serverless computing / L. Wang et al. *2024 International Symposium on Networks, Computers and Communications (ISNCC)* : Proccedings of the Intern. Conf., Washington, DC, USA, Oct. 22–25, 2024 / IEEE. Piscataway, NJ, USA : IEEE, 2024. P. 1–6. URL: https://doi.org/10.1109/ISNCC62547.2024.10759023.

134. Performance analysis of machine learning centered workload prediction models for cloud / D. Saxena et al. *IEEE Transactions on Parallel and Distributed Systems*. 2023. Vol. 34, № 4. P. 1313–1330. URL: https://doi.org/10.1109/TPDS.2023.3240567.

135. Lin C., Khazaei H. Modeling and Optimization of Performance and Cost of Serverless Applications. *IEEE Transactions on Parallel and Distributed Systems.* 2021. Vol. 32, № 3. P. 615–632. URL: https://doi.org/10.1109/TPDS.2020.3028841.

136. Wen Z., Wang Y., Liu F. StepConf: SLO-Aware Dynamic Resource Configuration for Serverless Function Workflows. *IEEE INFOCOM 2022 – IEEE Conference on Computer Communications* : Proccedings of the Intern. Conf., London, UK (online), May 2–5, 2022 / IEEE. Piscataway, NJ, USA : IEEE, 2022. P. 1868–1877. URL: https://doi.org/10.1109/INFOCOM48880.2022.9796962.

137. Hassan H. B., Barakat S. A., Sarhan Q. I. Survey on serverless computing. *Journal of Cloud Computing.* 2021. Vol. 10, Art. 39. URL: https://doi.org/10.1186/s13677-021-00253-7.

138. VFL-Chain: Bulletproofing Federated Learning in the V2X Environments / A. Smahi et al. *Future Generation Computer Systems*. 2024. Vol. 155. P. 419–436. URL: https://doi.org/10.1016/j.future.2024.02.012.

139. Owl: Performance-Aware Scheduling for Resource-Efficient Function-as-a-Service Cloud / H. Tian et al. *SoCC'22 :* Proceedings of the 13th Symposium on Cloud Computing **,** San Francisco, CA, USA, Nov. 7–10, 2022 / Association for Computing Machinery. New York, NY, USA : ACM, 2022. P. 78–93. URL: https://doi.org/10.1145/3542929.3563470.

140. Atar R. Central limit theorem for a many-server queue with random service rates. *Annals of Applied Probability*. 2008. Vol. 18, № 4. P. 1548–1568. URL: https://doi.org/10.1214/07-AAP497.

141. Lalchandani A. P. Some Limit Theorems in Queuing Theory. Technical Report No. 29. New York : Cornell University, 1967. 82 p.

142. Whitt W. Approximations for the *GI/G/m* Queue. *Production and Operations Management.* 1993. Vol. 2, Issue. 2. P. 114–161. URL: https://doi.org/10.1111/j.1937-5956.1993.tb00094.x.

143. Grassmann W. K. Finding the Right Number of Servers in Real-World Queuing Systems. *INFORMS Journal on Applied Analytics*. 1988. Vol. 18, Issue 2. P. 94–104. URL: https://doi.org/10.1287/inte.18.2.94.

144. Pardo M. J., de la Fuente D. Optimal selection of the service rate for a finite input source fuzzy queuing system. *Fuzzy Sets and Systems*. 2008. Vol. 159, Issue. 3. P. 325–342. URL: https://doi.org/10.1016/j.fss.2007.05.014.

145. Yajima M., Phung-Duc T. A Central limit theorem for a Markov-modulated infinite-server queue with batch Poisson arrivals and binomial catastrophes. *Performance Evaluation.* 2019. Vol. 129. P. 2–14. URL: https://doi.org/10.1016/j.peva.2018.10.002.

146. Gordon W. J., Newell G. F. Closed Queuing Systems with Exponential Servers. *Operations Research*. 1967. Vol. 15, № 2. P. 254–265. URL: https://doi.org/10.1287/opre.15.2.254.

147. Ke J.-C., Lee S.-L., Liou C.-H. Machine Repair Problem in Production Systems with Spares and Server Vacations. *RAIRO – Operations Research*. 2009. Vol. 43, № 1. P. 35 – 54. URL: https://doi.org/10.1051/ro/2009004.

148. Nair A. N., Jacob M. J., Krishnamoorthy A. The Multi-Server M/M/(s, S) Queueing Inventory System. *Annal of Operations Research*. 2015. Vol. 233. P. 321–333. URL: https://doi.org/10.1007/s10479-013-1405-5.

149. Lebedev E., Livinska H. Multi-channel Queueing Networks with Input Flow Controlled by Semi-Markov Process. *Mathematics in Computer Science.* 2019. Vol. 13. P. 333–340. URL: https://doi.org/10.1007/s11786-019-00398-4.

150. Ahn H., Duenyas I., Lewis M. E. Optimal control of a two-stage tandem queuing system with flexible servers. *Probability in the Engineering and Informational Sciences*. 2002. Vol. 16, Issue 4. P. 453–469. URL: https://doi.org/10.1017/S0269964802164047.

151. Cao J., Li K., Stojmenovic I. Optimal Power Allocation and Load Distribution for Multiple Heterogeneous Multicore Server Processors across Clouds and Data Centers. *IEEE Transactions on Computers*. 2014. Vol. 63, Issue 1. P. 45–58. URL: https://doi.org/10.1109/TC.2013.122.

152. Massey W. A., Whitt W. Networks of infinite-server queues with nonstationary Poisson input. *Queueing Systems*. 1993. Vol. 13. P. 183–250. URL: https://doi.org/10.1007/BF01158933.

153. Кириченко О., Малик І., Кириченко О. Оцінки довжини черги в неоднорідних системах масового обслуговування зі змінними режимами роботи. *Scientific Progress: Theories, Applications and Global Impact* : Proceedings of the 3rd International Scientific and Practical Conference, Braga, Portugal, March 2-4, 2026 / European Open Science Space. Bratislava, Slovakia : EOSS, 2026. P. 258–260.

154. Koroliuk V. S., Limnios N. Stochastic Systems in Merging Phase Space. Singapore: World Scientific, 2005. 348 p. URL: https://doi.org/10.1142/5979.

155. Limnios N., Swishchuk A. Weak Convergence of DTSMRE in Series Scheme. In: Discrete-Time Semi-Markov Random Evolutions and Their Applications. Probability and Its Applications. Birkhäuser, Cham. 2025. P. 55–58. https://doi.org/10.1007/978-3-031-33429-0_4.

156. Limnios N., Wu B. Diffusion Approximation of Loss Queueing Systems. *Methodology and Computing in Applied Probability*. 2025. Vol. 27, Art. 20. URL: https://doi.org/10.1007/s11009-025-10141-1.

157. Fischer W., Meier-Hellstern K. The Markov-modulated Poisson process (MMPP) cookbook. *Performance Evaluation*. 1993. Vol. 18, Issue 2. P. 149–171. URL: https://doi.org/10.1016/0166-5316(93)90035-S.

158. FaaSPR: Latency-Oriented Placement and Routing Optimization for Serverless Workflow Processing / Y. Jia et al. *IEEE Transactions on Networking.* 2025. Vol. 33, № 4. P. 2063–2078. URL: https://doi.org/10.1109/TON.2025.3552407.

159. Hui X., Xu Y., Shen X. Exploring Function Granularity for Serverless Machine Learning Application with GPU Sharing. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*. 2025. Vol. 9, № 1, Art. 6. P. 1–28. URL: https://doi.org/10.1145/3711699.

160. Trident: A Provider-Oriented Resource Management Framework for Serverless Computing Platforms / B. Zhu et al. *IEEE Transactions on Services Computing*. 2025. Vol. 18, Issue 5. P. 3334–3347. URL: https://doi.org/10.1109/TSC.2025.3603867.

161. RUSH: Rule-Based Scheduling for Low-Latency Serverless Computing / P. A. Birajdar et al. *IEEE Networking Letters*. 2025. Vol. 8. P. 1–4. URL: https://doi.org/10.1109/LNET.2025.3603863.

162. Sustainable serverless computing with cold-start optimization and automatic workflow resource scheduling / S. Pan et al. *IEEE Transactions on*

Sustainable Computing. 2024. Vol. 9, № 3. P. 329–340. URL: https://doi.org/10.1109/TSUSC.2023.3311197.

163. Yu L., Fu L., Chenhao S. Serverless cold start performance optimization based on multi-request processing and adaptive hierarchical scaling. IEEE Access. 2024. Vol. 12. P. 136248–136262. URL: https://doi.org/10.1109/ACCESS.2024.3462389.

164. Hogg R. V., McKean J. W., Craig A. T. Introduction to mathematical statistics. 8th ed. Boston : Pearson, 2019. 746 p.

165. Joe H., Zhu R. Generalized Poisson distribution: the Property of Mixture of Poisson and Comparison With Negative Binomial Distribution. *Biometrical Journal.* 2005. Vol. 47, № 2. P. 219–229. URL: https://doi.org/10.1002/bimj.200410102.

166. Remling C. Spectral analysis of higher-order differential operators II: Fourth-order equations. *Journal of the London Mathematical Society*. 1999. Vol. 59. P. 188–206. URL: https://doi.org/10.1112/S0024610799007012.

167. A Sparse Function Prediction Approach for Cold Start Optimization and User Satisfaction Guarantee in Serverless / W. Zhang et al. *IEEE Transactions on Parallel and Distributed Systems*. 2025. Vol. 36, Issue 11. P. 2198–2213. URL: https://doi.org/10.1109/TPDS.2025.3602440.

168. Available CloudWatch metrics for Amazon SQS. *AWS* : website. URL: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-available-cloudwatch-metrics.html (дата звернення: 26.01.2026).

169. Кириченко О., Остапов С., Кириченко О. Безсерверний фреймворк із прогнозним масштабуванням на основі DeepAR для розподілених обчислень. *Trends and Prospects for the Development of Science and Education* : Proceedings of the 2nd International Scientific Conference, Oxford, United Kingdom, 10 July 2025 / Lulu Press, Inc. Morrisville, NC, USA : Lulu Press, 2025. P. 159–162. URL: https://doi.org/10.64076/iedc250710.24.

170. DeepAR: Probabilistic forecasting with autoregressive recurrent networks / D. Salinas et al. *International Journal of Forecasting.* 2020. Vol. 36, Issue. 3. P. 1181–1191. URL: https://doi.org/10.1016/j.ijforecast.2019.07.001.

171. Use the SageMaker AI DeepAR forecasting algorithm. *AWS* : website. URL: https://docs.aws.amazon.com/sagemaker/latest/dg/deepar.html (дата звернення: 27.01.2026).

172. Now available in Amazon SageMaker: DeepAR algorithm for more accurate time series forecasting / T. Januschowski et al. *AWS* : URL: https://aws.amazon.com/blogs/machine-learning/now-available-in-amazon-sagemaker-deepar-algorithm-for-more-accurate-time-series-forecasting/ (дата звернення: 27.01.2026).

173. Heyman J., Holmberg L., Baldwin A. What is Locust? *Locust* : URL: https://docs.locust.io/en/stable/what-is-locust.html (дата звернення: 27.01.2026).

174. Cao J., Li K., Stojmenovic I. Optimal Power Allocation and Load Distribution for Multiple Heterogeneous Multicore Server Processors Across Clouds and Data Centers. *IEEE Transactions on Computers*. 2014. Vol. 63, Issue 1. P. 45–58. URL: https://doi.org/10.1109/TC.2013.122.

175. Kumar N. S., Samy S. S. Cold Start Prediction and Provisioning Optimization in Serverless Computing Using Deep Learning. *Concurrency and Computation: Practice and Experience.* 2025. Vol. 37, № 4–5. URL: https://doi.org/10.1002/cpe.8392.

176. Array Programming with NumPy / C. R. Harris et al. *Nature.* 2020. Vol. 585. P. 357–362. URL: https://doi.org/10.1038/s41586-020-2649-2.

177. SciPy 1.0: fundamental algorithms for scientific computing in Python / P. Virtanen et al. *Nature Methods*. 2020. Vol. 17. P. 261–272. URL: https://doi.org/10.1038/s41592-019-0686-2.

178. Hunter J. D. Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering.* 2007. Vol. 9, № 3. P. 90–95. URL: https://doi.org/10.1109/MCSE.2007.55.

179. Scikit-learn: Machine Learning in Python / F. Pedregosa et al. *Journal of Machine Learning Research*. 2011. Vol. 12. P. 2825–2830. URL: https://jmlr.org/papers/v12/pedregosa11a.html.

180. Why the Monte Carlo method is so important today / D. P. Kroese et al. *WIREs Computational Statistics*. 2014. Vol. 6, Issue 6. P. 386–392. URL: https://doi.org/10.1002/wics.1314.

181. Cloud-Based serverless computing enables accelerated Monte Carlo simulations for nuclear medicine imaging / R. Bayerlein et al. *Biomedical Physics & Engineering Express.* 2024. Vol. 10, № 4. Art. 045053. URL: https://doi.org/10.1088/2057-1976/ad5847.

182. Monte Carlo Simulation-Based Robust Workflow Scheduling for Spot Instances in Cloud Environments / Q. Wu et al. *Tsinghua Science and Technology.* 2024. Vol. 29, № 1. P. 112–126. URL: https://doi.org/10.26599/TST.2022.9010065.

183. Zhou F. Advances in Temporal Point Processes: Bayesian, Neural, and LLM Approaches. *ArXiv preprint arXiv:2501.14291v2.* 2025. URL: https://doi.org/10.48550/arXiv.2501.14291.

184. Reynolds D. Gaussian Mixture Models. *Encyclopedia of Biometrics* / ed. by S. Z. Li, A. Jain. Boston, MA : Springer, 2009. P. 659–663. URL: https://doi.org/10.1007/978-0-387-73003-5_196.

185. McLachlan G. J., Lee S. X., Rathnayake S. I. Finite Mixture Models. *Annual Review of Statistics and Its Application*. 2019. Vol. 6. P. 355–378. URL: https://doi.org/10.1146/annurev-statistics-031017-100325.

186. Neath A. A., Cavanaugh J. E. The Bayesian information criterion: background, derivation, and applications. *WIREs Computational Statistics*. 2012. Vol. 4, Issue 2. P. 199–203. URL: https://doi.org/10.1002/wics.199.

187. Berger V. W., Zhou Y. Kolmogorov–Smirnov Test: Overview. *Wiley StatsRef: Statistics Reference Online*. 2014. URL: https://doi.org/10.1002/9781118445112.stat06558.

188. Simard R., L'Ecuyer P. Computing the Two-Sided Kolmogorov-Smirnov Distribution. *Journal of Statistical Software*. 2011. Vol. 39, Issue 11. P. 1–18. URL: https://doi.org/10.18637/jss.v039.i11.

189. Basu A., Shioya H., Park, C. Statistical Inference: The Minimum Distance Approach. 1st ed. New York : Chapman and Hall/CRC, 2011. 429 p. URL: https://doi.org/10.1201/b10956.

190. On Approximating Total Variation Distance / A. Bhattacharyya et al. *IJCAI-2023* : Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, Macao, China, Aug. 19–25, 2023 / IJCAI. California, USA : IJCAI, 2023. P. 3479–3487. URL: https://doi.org/10.24963/ijcai.2023/387.

191. Hirata M. A formalization of the Lévy–Prokhorov metric in Isabelle/HOL. Proceedings of the 15th International Conference on Interactive Theorem Proving (ITP 2024). Leibniz International Proceedings in Informatics (LIPIcs). 2024. Vol. 309. P. 21:1–21:18.
URL: https://doi.org/10.4230/LIPIcs.ITP.2024.21.

192. Kim T. K. T test as a Parametric Statistic. *Korean Journal of Anesthesiology.* 2015. Vol. 68, № 6. P. 540–546. URL: https://doi.org/10.4097/kjae.2015.68.6.540.

193. de Winter J. Using the Student's t-test with extremely small sample sizes. *Practical Assessment, Research, and Evaluation*. 2013. Vol. 18, № 1. Art. 10. URL: https://doi.org/10.7275/e4r6-dj05.

194. Chicco D., Sichenze A., Jurman G. A simple guide to the use of Student's t-test, Mann-Whitney U test, Chi-squared test, and Kruskal-Wallis test in biostatistics. *BioData Mining.* 2025. Vol. 18. Art. 56. URL: https://doi.org/10.1186/s13040-025-00465-6

ДОДАТКИ

Додаток А

СПИСОК ПУБЛІКАЦІЙ ЗА ТЕМОЮ ДИСЕРТАЦІЇ

Наукові праці, в яких опубліковані
основні наукові результати дисертації :

Наукові праці у виданнях, включених до переліку наукових фахових видань України та проіндексованих у наукометричній базі даних Scopus:

1. **Kyrychenko O.,** Ostapov S., Kyrychenko O. Design of a framework for serverless distributed data processing using queues. *Eastern-European Journal of Enterprise Technologies*. 2025. Vol.4, №9. P. 19–25. (**Scopus**) URL: https://doi.org/10.15587/1729-4061.2025.335723 (*Особистий внесок: Кириченко О. О. – теоретична та практична розробка положень, розробка фреймворку, проведення дослідження, написання; Остапов С. Е. – постановка задачі, визначення загальної схеми дослідження, обговорення результатів; Кириченко О. Л. – аналіз літературних джерел, візуалізація, обговорення результатів*)

Наукові праці у періодичних наукових виданнях, проіндексованих у наукометричній базі даних Scopus:

2. **Kyrychenko O. O.**, Ostapov S. E., Kyrychenko O. L. Optimization of SQS Configurations for Efficient Batch Data Processing. *WSEAS Transactions on Systems.* 2025. Vol. 24. P. 36–43. (**Scopus**) URL: https://doi.org/10.37394/23202.2025.24.4 (*Особистий внесок: Кириченко О. О. – проведення дослідження, розробка хмарного прототипу, огляд літератури, написання, редагування рукопису; Остапов С. Е. – постановка задачі, концептуалізація; Кириченко О. Л. – валідація результатів дослідження, аналіз літературних джерел, написання, редагування*)

Наукові праці у виданнях,

включених до переліку наукових фахових видань України:

3. Кириченко О., **Кириченко О**. Кешування даних у додатках з використанням безсерверної архітектури. *Information Technology: Computer Science, Software Engineering and Cyber Security*. 2024. №2. С. 42–49. URL: https://doi.org/10.32782/IT/2024-2-6 (*Особистий внесок: Кириченко О. О. – проведення дослідження, написання, редагування рукопису; Кириченко О. Л. – постановка задачі, керівництво дослідженням, рецензування та редагування*).

4. Кириченко О. Л., **Кириченко О. О.** Використання AWS APPSYNC для комунікації вебдодатків у реальному часі. *Вчені записки Таврійського національного університету імені В.І. Вернадського.* Серія : Технічні науки. 2024. 35(74), №4. С. 105–110. URL: https://doi.org/10.32782/2663-5941/2024.4/16 (*Особистий внесок: Кириченко О. О. – огляд літературних джерел, проведення дослідження, написання, редагування рукопису; Кириченко О. Л. – постановка задачі, визначення загальної схеми досліджень, обговорення результату*).

Наукові праці, які засвідчують апробацію матеріалів дисертації:

5. Антонюк Д. В., **Кириченко О. О.** Пакетна обробка даних на безсерверній архітектурі. *Проблеми інформатики та комп'ютерної техніки (ПІКТ–2023)* : праці XII праці Міжнар. наук.-практ. конф., м. Чернівці, 10–12 листопада 2023 р. Чернівці: Черн. нац. ун-т, 2023. С. 106–109. (*Особистий внесок: Кириченко О. О. – постановка задачі, методологія дослідження, проведення дослідження; Антонюк Д. В. – тестування, обговорення результатів, написання*).

6. Кириченко О. Л., **Кириченко О. О.** Покращення швидкодії безсерверних додатків за допомогою кешування. *Problems of Science and Technology: the Search for Innovative Solutions :* Proceedings of the XXIII International scientific and practical conference, Munich, Germany, 15–17 May,

2024. International Scientific Unity, 2024. P. 65–67. (*Особистий внесок: Кириченко О. О. – огляд літературних джерел, проведення досліджень, написання; Кириченко О. Л. – постановка задачі, редагування, рецензування)*.

7. **Kyrychenko O.**, Kyrychenko O. Real-time communication tools for web applications in a cloud environment. *The 13 th International Conference on Electronics, Communications and Computing's (IC ECCO)* : Materials of the Intern. Conf., Chisinau, Moldova, 17–18 October, 2024. P. 126–127. (*Особистий внесок: Кириченко О. О. – огляд літературних джерел, написання, рецензування та редагування; Кириченко О. Л. – постановка задачі, рецензування*).

8. **Кириченко О.О.,** Кириченко О. Л., Остапов С.Е. Аналіз продуктивності AWS SQS в умовах високих навантажень. *Проблеми інформатики та комп'ютерної техніки (ПІКТ–2024)* : праці XIII Міжнар. наук.-практ. конф., м. Чернівці, 01–03 листопада 2024 р. Чернівці : Черн. нац. ун-т, 2024. С. 42–44. (*Особистий внесок: Кириченко О. О. – проведення досліджень, аналіз та опис результатів; Кириченко О. Л. – рецензування, обговорення результатів; Остапов С. Е. – постановка задачі, обговорення результатів*).

9. **Кириченко О**., Остапов С., Кириченко О. Безсерверний фреймворк із прогнозним масштабуванням на основі DeepAR для розподілених обчислень. *Trends and Prospects for the Development of Science and Education* : Proceedings of the 2nd International Scientific Conference (Oxford, United Kingdom, 10 July 2025). Lulu Press, Inc., 2025. P. 159–162. (*Особистий внесок: Кириченко О. О. – методологія дослідження, розробка програмного забезпечення, дослідження, написання; Остапов С. Е. – постановка задачі, загальне керівництво; Кириченко О. Л. – тестування, обговорення результатів*).

10. **Кириченко О.**, Малик І., Кириченко О. Оцінки довжини черги в неоднорідних системах масового обслуговування зі змінними режимами роботи. *Scientific Progress: Theories, Applications and Global Impact* :

Proceedings of the 3rd International Scientific and Practical Conference (Braga, Portugal, March 2-4, 2026). European Open Science Space, 2026. P. 258–260. (*Особистий внесок: Кириченко О. О – аналіз та опис результату, написання; Малик І. В. – концептуалізація дослідження, аналіз результатів; Кириченко О. Л. – обговорення результатів, рецензування*).

Наукові праці, які додатково відображають наукові результати дисертації:

11. **Kyrychenko O.**, Ostapov S., Kyrychenko O. L. Predictive autoscaling in AWS Serverless by means of machine learning and SQS metrics. *CEUR Workshop Proceedings* : 6th International Workshop on Intelligent Information Technologies & Systems of Information Security (IntelITSIS 2025, April 4, 2025). 2025. Vol.-3963. P. 77–87. **(Scopus)** URL: https://ceur-ws.org/Vol-3963/ (*Особистий внесок: Кириченко О. О. – проведення дослідження, тестування, написання; Остапов С. Е. – постановка задачі, концептуалізація; Кириченко О. Л. – обговорення результатів, рецензування*).

Додаток Б

Відомості про апробацію результатів дисертації

Основні положення та наукові результати дисертаційної роботи доповідалися та обговорювалися на наукових семінарах кафедри програмного забезпечення комп'ютерних систем та кафедри математичних проблем управління і кібернетики Навчально-наукового інституту фізико-технічних та комп’ютерних наук Чернівецького національного університету імені Юрія Федьковича, а також на Всеукраїнських та Міжнародних наукових і науково-практичних конференціях:

– XII Міжнародна науково-практична конференція «Проблеми інформатики та комп’ютерної техніки (ПІКТ – 2023)», 10-12 листопада 2023 року, Чернівці, Україна (очна участь);

– XIII Міжнародна науково-практична конференція «Проблеми інформатики та комп’ютерної техніки (ПІКТ – 2024)», 01-03 листопада 2024 року, Чернівці, Україна (очна участь);

– The XXIII International scientific and practical conference «Problems of Science and Technology: the Search for Innovative Solutions», 15-17 May 2024, Munich, Germany (заочна участь);

– The 13th International Conference on Electronics, Communications and Computing's (IC ECCO), 17-18 October 2024, Chisinau, Moldova (онлайн доповідь);

– The 2nd International Scientific Conference «Trends and Prospects for the Development of Science and Education», 10 July 2025, Oxford, United Kingdom (заочна участь);

– The 6th International Workshop on Intelligent Information Technologies & Systems of Information Security (IntelITSIS 2025), 4 April 2025, Khmelnytskyi, Ukraine (онлайн доповідь);

– The 3rd International Scientific and Practical Conference «Scientific Progress: Theories, Applications and Global Impact», 2-4 March 2026, Braga, Portugal (заочна участь).

Додаток В

Акти впровадження результатів дисертаційної роботи

Bezoekadres
Zilverparkkade 86,
8232 WK Lelystad Nederland

Act

implementation of the results of the dissertation work
Oleksandr Kyrychenko
on the topic "Optimization of serverless computing in cloud environments",
submitted to the defense for obtaining a scientific degree Doctor of Philosophy
in the field of knowledge 12 - "Information technologies"
in specialty 121 - "Software engineering"

This act confirms that Finker Finance B.V. has evaluated and adopted for beta testing the results implemented in the dissertation research of Oleksandr Kyrychenko, a Ph.D. candidate of the Yuriy Fedkovych Chernivtsi National University, in particular some functionalities have been developed on the basis of the proposed framework.

The main advantages of the solutions developed by the applicant are:

- the ability to obtain analytical guarantees for service quality (SLA) through explicit mathematical formulas, rather than relying solely on empirical tuning;
- low computational cost of the optimization procedure compared to deep learning-based alternatives, as the approach relies on analytical formulas that can be evaluated in real time.

The solution has successfully completed beta testing and Finker Finance B.V. intends to incorporate it into its production cloud infrastructure for distributed data processing workflows.

17.03.2026

Joost van der Toorn
Finker Finance B.V.

АКТ
впровадження результатів дисертаційної роботи
Кириченка Олександра Олексійовича
на тему «Оптимізація безсерверних обчислень у хмарних середовищах»,
поданої до захисту для здобуття наукового ступеня доктора філософії
в галузі знань 12 – «Інформаційні технології»
за спеціальністю 121 – «Інженерія програмного забезпечення»

Цим актом підтверджується, що ФОП Вербицька Світлана Іванівна прийняла до використання результати, отримані в дисертаційному дослідженні здобувача наукового ступеня доктора філософії Чернівецького національного університету імені Юрія Федьковича Кириченка Олександра Олексійовича, а саме:

- розгорнуто багаторівневий фреймворк, що забезпечує обробку телеметричних даних від GPS-трекерів (координати, швидкість, стан датчиків, рівень палива) в режимі реального часу;
- інтегровано модуль прогнозного автомасштабування, що дозволяє заздалегідь розгортати обчислювальні ресурси у хмарному середовищі перед ранковими та вечірніми піками навантаження, коли кількість активних транспортних засобів різко зростає.

Основними перевагами впроваджених рішень для діяльності компанії є:

- можливість аналітично гарантувати якість обробки телеметричних даних (SLA) на основі явних математичних формул, що є критичним для забезпечення безперервного моніторингу транспортних засобів;
- суттєве зменшення затримок при обробці даних від GPS-трекерів завдяки проактивному масштабуванню ресурсів перед прогнозованими піками навантаження;
- безперешкодна інтеграція з існуючою хмарною інфраструктурою компанії на AWS без необхідності зміни коду додатків.

Рішення успішно пройшло бета-тестування та ФОП Вербицька С.І. використовує його у виробничій хмарній інфраструктурі для обробки телеметричних даних від GPS-трекерів транспортних засобів.

«20» квітня 2026 року

ПІДПРИЄМЕЦЬ Вербицька Світлана Іванівна 3066409129 Україна Чернівці

Керівник Вербицька С.І.

Проректор з наукової роботи
Чернівецького національного
університету імені Юрія Федьковича
доктор хімічних наук, доцент

_____ Юрій ХАЛАВКА

«02» квітня 2026 р.

Директор Навчально-наукового
інституту фізико-технічних та
комп'ютерних наук Чернівецького
національного університету імені
Юрія Федьковича
канд. техн. наук, доцент

_____ Петро ШПАТАР

«02» квітня 2026 р.

АКТ
впровадження результатів дисертаційної роботи
Кириченка Олександра Олексійовича «Оптимізація безсерверних обчислень у хмарних середовищах»
до захисту на здобуття наукового ступеня
доктора філософії за спеціальністю «Інженерія програмного забезпечення» в навчальному процесі

Комісія Чернівецького національного університету імені Юрія Федьковича в складі:

- заступник директора ННІФТКН з наукової роботи, д.ф.-м.н., професор Дуболазов О.В.;
- завідувача кафедри математичних проблем управління і кібернетики, д.ф.-м.н., професора Малика І.В.;
- завідувача кафедри програмного забезпечення комп'ютерних систем, доктора філософії, доцента Газдюк К.П.

склали цей акт про те, що результати дисертаційної роботи здобувача наукового ступеня доктора філософії за спеціальністю «Інженерія програмного забезпечення» Кириченко О.О. впроваджені в навчальний процес кафедр МПУіК та ПЗКС, зокрема, в дисциплінах «Безсерверні

обчислення у хмарних середовищах», «Проектування інформаційних систем».

При викладанні цих дисциплін використовуються наступні матеріали досліджень, які отримані в дисертаційній роботі:

- Реалізація архітектури інформаційної технології на основі системи черг повідомлень AWS SQS, безсерверних функцій AWS Lambda та сервісу моніторингу AWS CloudWatch. Програмне забезпечення демонструє принципи слабкої зв'язності компонентів інтелектуальної системи, подієво-орієнтованого виконання та автоматичного масштабування обчислювальних ресурсів на хмарній платформі.
- Реалізація модуля прогнозного автоматичного масштабування на основі нейронних мереж DeepAR. Модуль демонструє застосування методів машинного навчання та штучного інтелекту для оптимізації інфраструктурних рішень.
- Методичні матеріали з проектування розподілених інформаційних систем на безсерверній архітектурі. Матеріали включають порівняльний аналіз шаблонів комунікації в безсерверних системах, критерії вибору архітектурних рішень для інтелектуальних систем з різними типами навантаження, а також практичні рекомендації щодо оптимізації конфігурації безсерверних компонентів на хмарній платформі AWS.

Використання зазначених результатів дозволило підвищити якість навчального процесу із згаданих дисциплін.

Заступник директора ННІФТКН з
наукової роботи д.ф.-м.н., професор О.В. Дуболазов

Завідувач кафедри МПУіК,
д.ф.-м.н., професор І.В. Малик

Завідувач кафедри ПЗКС,
доктор філософії, доцент К.П. Газдюк

Додаток Г

ЛІСТИНГ ЧАСТИНИ КОДУ ПРОГРАМИ

```
// Вміст файлу config.py
"""
Конфігурація для алгоритму оцінки параметрів системи масового обслуговування.
"""
from dataclasses import dataclass, field
from typing import List
import os


@dataclass
class ModelingConfig:
    """
    Конфігурація параметрів для modeling pipeline.

    Attributes:
        data_path: Шлях до CSV файлу з даними
        min_tasks_per_interval: Мінімальна кількість задач на інтервал (m)
        distribution_type: Тип розподілу ('gaussian' або 'shifted_exponential')
        max_components: Максимальна кількість компонентів суміші (K)
        em_max_iter: Максимальна кількість ітерацій EM алгоритму
        em_tolerance: Толерантність для збіжності EM
        ks_alpha: Рівень значущості для тесту Колмогорова-Смірнова
        num_servers: Кількість серверів у системі (C)
        server_capacities: Інтенсивності обробки для серверів (μ_1, ..., μ_C)
        simulation_duration: Час симуляції системи (T)
        monte_carlo_runs: Кількість прогонів Monte Carlo
        confidence_level: Рівень довіри для довірчих інтервалів
        output_dir: Директорія для збереження результатів
        save_intermediate: Зберігати проміжні результати
        verbose: Виводити детальну інформацію під час виконання
        random_seed: Seed для генератора випадкових чисел (None = випадковий)
    """

    # Дані
    data_path: str = "../converted_data.csv"

    # Крок 1: Розбиття на інтервали
    min_tasks_per_interval: int = 30

    # Крок 2: Оцінка параметрів суміші
    distribution_type: str = 'gaussian'  # 'gaussian' або 'shifted_exponential'
    max_components: int = 5
    em_max_iter: int = 100
    em_tolerance: float = 1e-6

    # Крок 3: Тест Колмогорова-Смірнова
    ks_alpha: float = 0.05

    # Крок 5: Симуляція СМО
    num_servers: int = 3
    server_capacities: List[float] = field(default_factory=lambda: [0.5, 1.0, 1.5])
    simulation_duration: float = 3600.0  # секунди
    intensity_multiplier: float = 1.0  # Множник для збільшення інтенсивності надходження
```

```
max_queue_size: int = None  # Максимальний розмір черги (None = необмежена)

# Крок 6: Monte Carlo аналіз
monte_carlo_runs: int = 100
confidence_level: float = 0.95

# Виведення
output_dir: str = "modeling_results"
save_intermediate: bool = True
verbose: bool = True

# Випадковість
random_seed: int = None

def __post_init__(self):
    """Валідація параметрів після ініціалізації."""
    # Перевірка типу розподілу
    valid_distributions = ['gaussian', 'shifted_exponential']
    if self.distribution_type not in valid_distributions:
        raise ValueError(
            f"distribution_type має бути одним з {valid_distributions}, "
            f"отримано: {self.distribution_type}"
        )

    # Перевірка позитивних значень
    if self.min_tasks_per_interval <= 0:
        raise ValueError("min_tasks_per_interval має бути > 0")

    if self.max_components <= 0:
        raise ValueError("max_components має бути > 0")

    if self.num_servers <= 0:
        raise ValueError("num_servers має бути > 0")

    if len(self.server_capacities) != self.num_servers:
        raise ValueError(
            f"Кількість server_capacities ({len(self.server_capacities)}) "
            f"має дорівнювати num_servers ({self.num_servers})"
        )

    if any(cap <= 0 for cap in self.server_capacities):
        raise ValueError("Всі server_capacities мають бути > 0")

    if self.monte_carlo_runs <= 0:
        raise ValueError("monte_carlo_runs має бути > 0")

    if not (0 < self.confidence_level < 1):
        raise ValueError("confidence_level має бути між 0 і 1")

    if not (0 < self.ks_alpha < 1):
        raise ValueError("ks_alpha має бути між 0 і 1")

    if self.intensity_multiplier <= 0:
        raise ValueError("intensity_multiplier має бути > 0")

    # Створення директорії для результатів якщо не існує
    if not os.path.exists(self.output_dir):
```

```
        os.makedirs(self.output_dir, exist_ok=True)

    def get_absolute_data_path(self, base_dir: str = None) -> str:
        """
        Отримати абсолютний шлях до файлу даних.

        Args:
            base_dir: Базова директорія (якщо None, використовується поточна)

        Returns:
            Абсолютний шлях до файлу даних
        """
        if os.path.isabs(self.data_path):
            return self.data_path

        if base_dir is None:
            base_dir = os.getcwd()

        return os.path.abspath(os.path.join(base_dir, self.data_path))

    def to_dict(self) -> dict:
        """Конвертувати конфігурацію в словник."""
        return {
            'data_path': self.data_path,
            'min_tasks_per_interval': self.min_tasks_per_interval,
            'distribution_type': self.distribution_type,
            'max_components': self.max_components,
            'em_max_iter': self.em_max_iter,
            'em_tolerance': self.em_tolerance,
            'ks_alpha': self.ks_alpha,
            'num_servers': self.num_servers,
            'server_capacities': self.server_capacities,
            'simulation_duration': self.simulation_duration,
            'intensity_multiplier': self.intensity_multiplier,
            'max_queue_size': self.max_queue_size,
            'monte_carlo_runs': self.monte_carlo_runs,
            'confidence_level': self.confidence_level,
            'output_dir': self.output_dir,
            'save_intermediate': self.save_intermediate,
            'verbose': self.verbose,
            'random_seed': self.random_seed,
        }

    def __repr__(self) -> str:
        """Зручне представлення конфігурації."""
        return (
            f"ModelingConfig(\n"
            f"  data_path='{self.data_path}',\n"
            f"  min_tasks_per_interval={self.min_tasks_per_interval},\n"
            f"  distribution='{self.distribution_type}',\n"
            f"  max_components={self.max_components},\n"
            f"  servers={self.num_servers},\n"
            f"  mc_runs={self.monte_carlo_runs},\n"
            f"  output_dir='{self.output_dir}'\n"
            f")"
        )
```

```
// Вміст файлу data_loader.py
"""
Завантаження та валідація даних з CSV файлу.
"""
import pandas as pd
import numpy as np
from typing import Tuple, Dict
from datetime import datetime


class DataLoader:
    """
    Завантаження та первинна обробка даних для аналізу СМО.

    Attributes:
        filepath: Шлях до CSV файлу
        arrival_times: Часи надходження задач (відносні, в секундах)
        processing_times: Часи обробки задач (в секундах)
        metadata: Метадані про завантажені дані
        raw_data: Сирі дані з CSV (DataFrame)
    """

    def __init__(self, filepath: str = None):
        """
        Ініціалізація DataLoader.

        Args:
            filepath: Шлях до CSV файлу з даними
        """
        self.filepath = filepath
        self.arrival_times = None
        self.processing_times = None
        self.metadata = {}
        self.raw_data = None

    def load_from_csv(self, filepath: str = None) -> Tuple[np.ndarray, np.ndarray]:
        """
        Завантаження даних з CSV файлу.

        Очікуваний формат CSV:
        - timestamp: ISO 8601 timestamp
        - arrival_time: Unix timestamp в секундах
        - processing_time: Час обробки в секундах

        Args:
            filepath: Шлях до CSV файлу (перевизначає self.filepath)

        Returns:
            Tuple[np.ndarray, np.ndarray]: arrival_times, processing_times

        Raises:
            FileNotFoundError: Якщо файл не знайдено
            ValueError: Якщо структура файлу некоректна
        """
        if filepath is not None:
            self.filepath = filepath
```

```
    if self.filepath is None:
        raise ValueError("Filepath не вказано")

    # Читання CSV
    try:
        self.raw_data = pd.read_csv(self.filepath)
    except FileNotFoundError:
        raise FileNotFoundError(f"Файл не знайдено: {self.filepath}")
    except Exception as e:
        raise ValueError(f"Помилка читання CSV: {str(e)}")

    # Перевірка наявності необхідних колонок
    required_columns = ['timestamp', 'arrival_time', 'processing_time']
    missing_columns = set(required_columns) - set(self.raw_data.columns)
    if missing_columns:
        raise ValueError(
            f"Відсутні необхідні колонки: {missing_columns}. "
            f"Наявні колонки: {list(self.raw_data.columns)}"
        )

    # Валідація даних
    self.validate_data()

    # Конвертація даних
    self.arrival_times = self.raw_data['arrival_time'].values.copy()
    self.processing_times = self.raw_data['processing_time'].values.copy()

    # Сортування за arrival_time
    sort_indices = np.argsort(self.arrival_times)
    self.arrival_times = self.arrival_times[sort_indices]
    self.processing_times = self.processing_times[sort_indices]

    # Нормалізація: зробити arrival_times відносними (починаючи з 0)
    self.normalize_timestamps()

    # Збір метаданих
    self._collect_metadata()

    return self.arrival_times, self.processing_times

def validate_data(self) -> bool:
    """
    Валідація завантажених даних.

    Перевірки:
    - Відсутність NaN значень
    - Позитивні значення для processing_time
    - Коректність arrival_time

    Returns:
        bool: True якщо дані валідні

    Raises:
        ValueError: Якщо дані не проходять валідацію
    """
    if self.raw_data is None:
        raise ValueError("Дані не завантажені")
```

```
    # Перевірка на NaN
    if self.raw_data.isnull().any().any():
        nan_info = self.raw_data.isnull().sum()
        raise ValueError(f"Знайдено NaN значення:\n{nan_info[nan_info > 0]}")

    # Перевірка arrival_time
    if (self.raw_data['arrival_time'] < 0).any():
        raise ValueError("arrival_time містить негативні значення")

    # Перевірка processing_time
    if (self.raw_data['processing_time'] <= 0).any():
        raise ValueError("processing_time містить некоректні значення (<=0)")

    # Перевірка на дублікати
    duplicates = self.raw_data.duplicated().sum()
    if duplicates > 0:
        print(f"Попередження: Знайдено {duplicates} дублікатів. Вони будуть збережені.")

    return True

def normalize_timestamps(self) -> None:
    """
    Нормалізація arrival_times: зробити відносними від першого часу.

    Перетворює абсолютні Unix timestamps у відносні секунди від t_0.
    """
    if self.arrival_times is None:
        raise ValueError("arrival_times не завантажені")

    t_0 = self.arrival_times[0]
    self.arrival_times = self.arrival_times - t_0

def get_statistics(self) -> Dict:
    """
    Обчислення базової статистики по даних.

    Returns:
        Dict: Словник зі статистикою
    """
    if self.arrival_times is None or self.processing_times is None:
        raise ValueError("Дані не завантажені")

    stats = {
        'n_tasks': len(self.arrival_times),
        'arrival_time': {
            'min': float(self.arrival_times.min()),
            'max': float(self.arrival_times.max()),
            'mean': float(self.arrival_times.mean()),
            'std': float(self.arrival_times.std()),
            'range': float(self.arrival_times.max() - self.arrival_times.min()),
        },
        'processing_time': {
            'min': float(self.processing_times.min()),
            'max': float(self.processing_times.max()),
            'mean': float(self.processing_times.mean()),
            'std': float(self.processing_times.std()),
```

```
            'median': float(np.median(self.processing_times)),
            'q25': float(np.percentile(self.processing_times, 25)),
            'q75': float(np.percentile(self.processing_times, 75)),
        },
        'arrival_rate': {
            'overall': len(self.arrival_times) / (self.arrival_times.max() - self.arrival_times.min())
                    if self.arrival_times.max() > self.arrival_times.min() else 0,
        }
    }

    # Обчислення інтервалів між надходженнями
    if len(self.arrival_times) > 1:
        inter_arrival_times = np.diff(self.arrival_times)
        stats['inter_arrival_time'] = {
            'min': float(inter_arrival_times.min()),
            'max': float(inter_arrival_times.max()),
            'mean': float(inter_arrival_times.mean()),
            'std': float(inter_arrival_times.std()),
        }

    return stats

def _collect_metadata(self) -> None:
    """Збір метаданих про дані."""
    self.metadata = {
        'filepath': self.filepath,
        'n_records': len(self.raw_data),
        'load_time': datetime.now().isoformat(),
        'columns': list(self.raw_data.columns),
        'time_range_sec': float(self.arrival_times.max() - self.arrival_times.min())
                    if self.arrival_times is not None else 0,
    }

    # Додати інформацію про timestamps якщо є
    if 'timestamp' in self.raw_data.columns:
        try:
            timestamps = pd.to_datetime(self.raw_data['timestamp'])
            self.metadata['timestamp_start'] = str(timestamps.min())
            self.metadata['timestamp_end'] = str(timestamps.max())
        except:
            pass

def export_summary(self, output_path: str) -> None:
    """
    Експорт статистики та метаданих у текстовий файл.

    Args:
        output_path: Шлях для збереження звіту
    """
    if self.arrival_times is None:
        raise ValueError("Дані не завантажені")

    stats = self.get_statistics()

    with open(output_path, 'w', encoding='utf-8') as f:
        f.write("=" * 70 + "\n")
        f.write("ЗВІТ ПРО ЗАВАНТАЖЕННЯ ДАНИХ\n")
```

```
            f.write("=" * 70 + "\n\n")

            # Метадані
            f.write("МЕТАДАНІ:\n")
            f.write("-" * 70 + "\n")
            for key, value in self.metadata.items():
                f.write(f"{key}: {value}\n")

            # Статистика
            f.write("\n\nСТАТИСТИКА:\n")
            f.write("-" * 70 + "\n")
            f.write(f"Кількість задач: {stats['n_tasks']}\n\n")

            f.write("Час надходження (відносний, сек):\n")
            for key, value in stats['arrival_time'].items():
                f.write(f"  {key}: {value:.4f}\n")

            f.write("\nЧас обробки (сек):\n")
            for key, value in stats['processing_time'].items():
                f.write(f"  {key}: {value:.6f}\n")

            if 'inter_arrival_time' in stats:
                f.write("\nІнтервали між надходженнями (сек):\n")
                for key, value in stats['inter_arrival_time'].items():
                    f.write(f"  {key}: {value:.4f}\n")

            f.write("\nІнтенсивність надходження:\n")
            f.write(f"  загальна: {stats['arrival_rate']['overall']:.6f} задач/сек\n")
            f.write(f"  або: {stats['arrival_rate']['overall'] * 60:.4f} задач/хв\n")

        print(f"Звіт збережено: {output_path}")

    def __repr__(self) -> str:
        """Зручне представлення DataLoader."""
        if self.arrival_times is None:
            return f"DataLoader(filepath='{self.filepath}', data=Not loaded)"
        else:
            return (
                f"DataLoader(\n"
                f"  filepath='{self.filepath}',\n"
                f"  n_tasks={len(self.arrival_times)},\n"
                f"  time_range={self.arrival_times.max():.2f} sec\n"
                f")"
            )
```

```
// Вміст файлу interval_partitioning.py

"""
Розбиття часового ряду на інтервали з гарантією мінімальної кількості задач.
"""
import numpy as np
from typing import List, Tuple, Dict
import json


class IntervalPartitioner:
    """
```

```
Розбиття [t_MIN, t_MAX] на U підінтервалів з гарантією мінімальної кількості задач.

Алгоритм забезпечує:
- #{t_i: t_i ∈ [T_j, T_{j+1}]} = m для j=0,...,U-2
- #{t_i: t_i ∈ [T_{U-1}, T_U]} >= m

Attributes:
    min_tasks_per_interval: Мінімальна кількість задач на інтервал (m)
    intervals: Список інтервалів [(T_0, T_1), (T_1, T_2), ..., (T_{U-1}, T_U)]
    interval_stats: Статистика для кожного інтервалу
"""

def __init__(self, min_tasks_per_interval: int = 30):
    """
    Ініціалізація IntervalPartitioner.

    Args:
        min_tasks_per_interval: Мінімальна кількість задач на інтервал (m)

    Raises:
        ValueError: Якщо m <= 0
    """
    if min_tasks_per_interval <= 0:
        raise ValueError("min_tasks_per_interval має бути > 0")

    self.min_tasks_per_interval = min_tasks_per_interval
    self.intervals = []
    self.interval_stats = []

def create_intervals(self, arrival_times: np.ndarray) -> List[Tuple[float, float]]:
    """
    Створення інтервалів з гарантією мінімальної кількості задач.

    Алгоритм:
    1. Сортуємо arrival_times
    2. Ітеративно набираємо по m задач
    3. Створюємо інтервал [t_first, t_last]
    4. Перевіряємо останній інтервал (має бути >= m)

    Args:
        arrival_times: Часи надходження задач (відносні)

    Returns:
        List[Tuple[float, float]]: Список інтервалів

    Raises:
        ValueError: Якщо даних недостатньо для створення хоча б одного інтервалу
    """
    if len(arrival_times) < self.min_tasks_per_interval:
        raise ValueError(
            f"Недостатньо даних: {len(arrival_times)} задач < "
            f"min_tasks_per_interval ({self.min_tasks_per_interval})"
        )

    # Сортування
    sorted_times = np.sort(arrival_times)
```

```
    # Створення інтервалів
    intervals = []
    idx = 0
    n = len(sorted_times)

    while idx < n:
        # Взяти наступні m задач (або решту якщо залишилось менше 2*m)
        remaining = n - idx

        if remaining < 2 * self.min_tasks_per_interval:
            # Останній інтервал: беремо всі залишкові задачі
            interval_tasks = sorted_times[idx:]
            t_start = interval_tasks[0]
            t_end = interval_tasks[-1]
            intervals.append((t_start, t_end))
            break
        else:
            # Звичайний інтервал: беремо рівно m задач
            interval_tasks = sorted_times[idx:idx + self.min_tasks_per_interval]
            t_start = interval_tasks[0]
            t_end = interval_tasks[-1]
            intervals.append((t_start, t_end))
            idx += self.min_tasks_per_interval

    self.intervals = intervals
    self._compute_interval_stats(sorted_times)

    return self.intervals

def validate_partitioning(self, arrival_times: np.ndarray) -> bool:
    """
    Перевірка коректності розбиття.

    Перевірки:
    - Кожен інтервал (окрім останнього) має рівно m задач
    - Останній інтервал має >= m задач
    - Інтервали не перетинаються
    - Всі задачі покриті інтервалами

    Args:
        arrival_times: Часи надходження задач

    Returns:
        bool: True якщо розбиття валідне

    Raises:
        ValueError: Якщо розбиття не пройшло валідацію
    """
    if not self.intervals:
        raise ValueError("Інтервали не створені")

    sorted_times = np.sort(arrival_times)
    total_tasks_covered = 0

    for idx, (t_start, t_end) in enumerate(self.intervals):
        # Знайти задачі в інтервалі
        tasks_in_interval = sorted_times[
```

```
            (sorted_times >= t_start) & (sorted_times <= t_end)
        ]
        n_tasks = len(tasks_in_interval)

        # Перевірка кількості задач
        if idx < len(self.intervals) - 1:
            # Не останній інтервал: має бути >= m
            if n_tasks < self.min_tasks_per_interval:
                raise ValueError(
                    f"Інтервал {idx} має {n_tasks} задач < "
                    f"min={self.min_tasks_per_interval}"
                )
        else:
            # Останній інтервал: має бути >= m
            if n_tasks < self.min_tasks_per_interval:
                raise ValueError(
                    f"Останній інтервал має {n_tasks} задач < "
                    f"min={self.min_tasks_per_interval}"
                )

        total_tasks_covered += n_tasks

    # Перевірка що всі задачі покриті
    # Примітка: через перекриття границь може бути подвійний підрахунок
    # Перевіряємо лише що всі задачі в діапазоні [T_0, T_U]
    if len(sorted_times) > 0:
        if sorted_times[0] < self.intervals[0][0]:
            raise ValueError("Є задачі до першого інтервалу")
        if sorted_times[-1] > self.intervals[-1][1]:
            raise ValueError("Є задачі після останнього інтервалу")

    return True

def get_interval_for_time(self, t: float) -> int:
    """
    Знайти індекс інтервалу для заданого часу t.

    Args:
        t: Час

    Returns:
        int: Індекс інтервалу (0-based) або -1 якщо t поза всіма інтервалами
    """
    if not self.intervals:
        raise ValueError("Інтервали не створені")

    for idx, (t_start, t_end) in enumerate(self.intervals):
        if t_start <= t <= t_end:
            return idx

    return -1

def get_tasks_in_interval(
    self,
    interval_idx: int,
    arrival_times: np.ndarray,
    processing_times: np.ndarray = None
```

```
) -> Tuple[np.ndarray, np.ndarray]:
    """
    Отримати задачі для конкретного інтервалу.

    Args:
        interval_idx: Індекс інтервалу (0-based)
        arrival_times: Часи надходження задач
        processing_times: Часи обробки задач (опційно)

    Returns:
        Tuple[np.ndarray, np.ndarray]: arrival_times, processing_times для інтервалу

    Raises:
        IndexError: Якщо interval_idx поза межами
    """
    if not self.intervals:
        raise ValueError("Інтервали не створені")

    if interval_idx < 0 or interval_idx >= len(self.intervals):
        raise IndexError(f"Індекс інтервалу {interval_idx} поза межами [0, {len(self.intervals)-1}]")

    t_start, t_end = self.intervals[interval_idx]

    # Фільтрація задач
    mask = (arrival_times >= t_start) & (arrival_times <= t_end)
    interval_arrivals = arrival_times[mask]

    if processing_times is not None:
        interval_processing = processing_times[mask]
        return interval_arrivals, interval_processing
    else:
        return interval_arrivals, None

def _compute_interval_stats(self, sorted_times: np.ndarray) -> None:
    """
    Обчислення статистики для кожного інтервалу.

    Args:
        sorted_times: Відсортовані часи надходження
    """
    self.interval_stats = []

    for idx, (t_start, t_end) in enumerate(self.intervals):
        tasks_in_interval = sorted_times[
            (sorted_times >= t_start) & (sorted_times <= t_end)
        ]

        stats = {
            'interval_idx': idx,
            't_start': float(t_start),
            't_end': float(t_end),
            'duration': float(t_end - t_start),
            'n_tasks': len(tasks_in_interval),
            'arrival_rate': len(tasks_in_interval) / (t_end - t_start)
                            if t_end > t_start else 0,
        }
```

```
        # Інтервали між надходженнями в цьому інтервалі
        if len(tasks_in_interval) > 1:
            inter_arrivals = np.diff(tasks_in_interval)
            stats['inter_arrival_mean'] = float(np.mean(inter_arrivals))
            stats['inter_arrival_std'] = float(np.std(inter_arrivals))
        else:
            stats['inter_arrival_mean'] = None
            stats['inter_arrival_std'] = None

        self.interval_stats.append(stats)

def merge_small_intervals(
    self,
    arrival_times: np.ndarray,
    min_threshold: int = None
) -> List[Tuple[float, float]]:
    """
    Об'єднання інтервалів з малою кількістю задач.

    Args:
        arrival_times: Часи надходження задач
        min_threshold: Поріг для об'єднання (за замовчуванням = min_tasks_per_interval)

    Returns:
        List[Tuple[float, float]]: Оновлений список інтервалів
    """
    if not self.intervals:
        raise ValueError("Інтервали не створені")

    if min_threshold is None:
        min_threshold = self.min_tasks_per_interval

    sorted_times = np.sort(arrival_times)
    merged_intervals = []
    i = 0

    while i < len(self.intervals):
        current_start, current_end = self.intervals[i]

        # Знайти задачі в поточному інтервалі
        tasks = sorted_times[
            (sorted_times >= current_start) & (sorted_times <= current_end)
        ]

        # Якщо недостатньо задач і є наступний інтервал - об'єднуємо
        if len(tasks) < min_threshold and i < len(self.intervals) - 1:
            next_start, next_end = self.intervals[i + 1]
            merged_intervals.append((current_start, next_end))
            i += 2  # пропускаємо наступний інтервал
        else:
            merged_intervals.append((current_start, current_end))
            i += 1

    self.intervals = merged_intervals
    self._compute_interval_stats(sorted_times)

    return self.intervals
```

```
    def export_to_json(self, output_path: str) -> None:
        """
        Експорт інтервалів та статистики в JSON.

        Args:
            output_path: Шлях для збереження JSON файлу
        """
        if not self.intervals:
            raise ValueError("Інтервали не створені")

        data = {
            'min_tasks_per_interval': self.min_tasks_per_interval,
            'n_intervals': len(self.intervals),
            'intervals': [
                {'t_start': float(t_start), 't_end': float(t_end)}
                for t_start, t_end in self.intervals
            ],
            'interval_stats': self.interval_stats,
        }

        with open(output_path, 'w', encoding='utf-8') as f:
            json.dump(data, f, indent=2, ensure_ascii=False)

        print(f"Інтервали збережені: {output_path}")

    def __repr__(self) -> str:
        """Зручне представлення IntervalPartitioner."""
        if not self.intervals:
            return f"IntervalPartitioner(m={self.min_tasks_per_interval}, intervals=Not created)"
        else:
            return (
                f"IntervalPartitioner(\n"
                f"  m={self.min_tasks_per_interval},\n"
                f"  n_intervals={len(self.intervals)},\n"
                f"  time_range=[{self.intervals[0][0]:.2f}, {self.intervals[-1][1]:.2f}]\n"
                f")"
            )
```

// Вміст файлу mixture_estimator.py

```
"""
Оцінка параметрів сумішей розподілів з використанням EM алгоритму.
"""
import numpy as np
from dataclasses import dataclass
from typing import List, Tuple, Optional
from scipy.stats import norm
from sklearn.mixture import GaussianMixture as SKLearnGMM
import warnings


@dataclass
class GaussianMixtureParams:
    """
    Параметри гауссівської суміші.
```

```
f(x) = Σ w_i * N(x | μ_i, σ²_i)

Attributes:
    K: Кількість компонентів суміші
    weights: Ваги компонентів w = (w_1, ..., w_K), Σw_i = 1
    means: Середні значення μ = (μ_1, ..., μ_K)
    stds: Стандартні відхилення σ = (σ_1, ..., σ_K)
    covariances: Дисперсії σ² = (σ²_1, ..., σ²_K)
    bic_score: Байєсівський інформаційний критерій
    aic_score: Інформаційний критерій Акаїке
    log_likelihood: Логарифм правдоподібності
    n_iterations: Кількість ітерацій EM до збіжності
"""
K: int
weights: np.ndarray
means: np.ndarray
stds: np.ndarray
covariances: np.ndarray
bic_score: float
aic_score: float
log_likelihood: float
n_iterations: int = 0

def pdf(self, x: np.ndarray) -> np.ndarray:
    """
    Обчислення щільності ймовірності суміші.

    Args:
        x: Точки для обчислення PDF

    Returns:
        np.ndarray: Значення PDF в кожній точці
    """
    x = np.atleast_1d(x)
    pdf_values = np.zeros_like(x, dtype=float)

    for k in range(self.K):
        pdf_values += self.weights[k] * norm.pdf(x, loc=self.means[k], scale=self.stds[k])

    return pdf_values

def sample(self, n_samples: int, random_state: Optional[int] = None) -> np.ndarray:
    """
    Генерація вибірки з суміші.

    Args:
        n_samples: Кількість зразків
        random_state: Seed для генератора

    Returns:
        np.ndarray: Вибірка з суміші
    """
    rng = np.random.default_rng(random_state)

    # Вибір компонентів згідно з вагами
    components = rng.choice(self.K, size=n_samples, p=self.weights)
```

```
        # Генерація з відповідних компонентів
        samples = np.array([
            rng.normal(loc=self.means[k], scale=self.stds[k])
            for k in components
        ])

        return samples


@dataclass
class ShiftedExponentialParams:
    """
    Параметри зсунутого показникового розподілу.

    f(x) = Σ w_i * λ_i * e^(-λ_i(x-l_i)), x >= l_i

    Attributes:
        K: Кількість компонентів
        weights: Ваги w = (w_1, ..., w_K)
        lambdas: Параметри інтенсивності λ = (λ_1, ..., λ_K)
        shifts: Зсуви l = (l_1, ..., l_K)
        bic_score: BIC
        aic_score: AIC
        log_likelihood: Log-likelihood
    """
    K: int
    weights: np.ndarray
    lambdas: np.ndarray
    shifts: np.ndarray
    bic_score: float
    aic_score: float
    log_likelihood: float
    n_iterations: int = 0


class EMAlgorithm:
    """
    Реалізація Expectation-Maximization алгоритму для гауссівських сумішей.

    Алгоритм:
    1. E-step: обчислення posterior ймовірностей γ_ik
    2. M-step: оновлення параметрів w, μ, σ на основі γ
    3. Повтор до збіжності
    """

    def __init__(self, max_iter: int = 100, tolerance: float = 1e-6, verbose: bool = False):
        """
        Ініціалізація EM алгоритму.

        Args:
            max_iter: Максимальна кількість ітерацій
            tolerance: Толерантність для збіжності
            verbose: Виводити прогрес
        """
        self.max_iter = max_iter
        self.tolerance = tolerance
        self.verbose = verbose
```

```
def e_step(
    self,
    data: np.ndarray,
    weights: np.ndarray,
    means: np.ndarray,
    stds: np.ndarray
) -> np.ndarray:
    """
    E-step: обчислення posterior ймовірностей (responsibilities).

    γ_ik = w_k * N(x_i | μ_k, σ²_k) / Σ_j [w_j * N(x_i | μ_j, σ²_j)]

    Args:
        data: Дані (n_samples,)
        weights: Поточні ваги (K,)
        means: Поточні середні (K,)
        stds: Поточні стандартні відхилення (K,)

    Returns:
        np.ndarray: Responsibilities (n_samples, K)
    """
    n = len(data)
    K = len(weights)
    responsibilities = np.zeros((n, K))

    # Обчислення PDF для кожного компонента
    for k in range(K):
        responsibilities[:, k] = weights[k] * norm.pdf(data, loc=means[k], scale=stds[k])

    # Нормалізація
    row_sums = responsibilities.sum(axis=1, keepdims=True)
    row_sums[row_sums == 0] = 1e-10  # уникнення ділення на 0
    responsibilities /= row_sums

    return responsibilities

def m_step(
    self,
    data: np.ndarray,
    responsibilities: np.ndarray
) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
    """
    M-step: оновлення параметрів на основі responsibilities.

    w_k = (1/N) Σ_i γ_ik
    μ_k = Σ_i (γ_ik * x_i) / Σ_i γ_ik
    σ²_k = Σ_i (γ_ik * (x_i - μ_k)²) / Σ_i γ_ik

    Args:
        data: Дані (n_samples,)
        responsibilities: Responsibilities (n_samples, K)

    Returns:
        Tuple[np.ndarray, np.ndarray, np.ndarray]: weights, means, stds
    """
    n = len(data)
```

```
    K = responsibilities.shape[1]

    # Оновлення ваг
    n_k = responsibilities.sum(axis=0)
    weights = n_k / n

    # Оновлення середніх
    means = np.zeros(K)
    for k in range(K):
        means[k] = (responsibilities[:, k] * data).sum() / n_k[k]

    # Оновлення стандартних відхилень
    stds = np.zeros(K)
    for k in range(K):
        diff = data - means[k]
        stds[k] = np.sqrt((responsibilities[:, k] * diff ** 2).sum() / n_k[k])

    # Захист від вироджених компонентів
    stds = np.maximum(stds, 1e-6)

    return weights, means, stds

def compute_log_likelihood(
    self,
    data: np.ndarray,
    weights: np.ndarray,
    means: np.ndarray,
    stds: np.ndarray
) -> float:
    """
    Обчислення логарифму правдоподібності.

    log L = Σ_i log[Σ_k w_k * N(x_i | μ_k, σ²_k)]

    Args:
        data: Дані
        weights: Ваги
        means: Середні
        stds: Стандартні відхилення

    Returns:
        float: Log-likelihood
    """
    n = len(data)
    K = len(weights)

    # Обчислення суми для кожного зразка
    log_prob_sum = np.zeros(n)
    for k in range(K):
        log_prob_sum += weights[k] * norm.pdf(data, loc=means[k], scale=stds[k])

    # Уникнення log(0)
    log_prob_sum = np.maximum(log_prob_sum, 1e-10)

    return np.sum(np.log(log_prob_sum))

def fit(
```

```
    self,
    data: np.ndarray,
    K: int,
    init_weights: Optional[np.ndarray] = None,
    init_means: Optional[np.ndarray] = None,
    init_stds: Optional[np.ndarray] = None,
    random_state: Optional[int] = None
) -> GaussianMixtureParams:
    """
    Повний EM цикл для оцінки параметрів.

    Args:
        data: Дані для fit
        K: Кількість компонентів
        init_weights: Початкові ваги (опційно)
        init_means: Початкові середні (опційно)
        init_stds: Початкові стандартні відхилення (опційно)
        random_state: Seed для ініціалізації

    Returns:
        GaussianMixtureParams: Оцінені параметри
    """
    data = np.atleast_1d(data).flatten()
    n = len(data)

    # Ініціалізація параметрів
    if init_weights is None or init_means is None or init_stds is None:
        weights, means, stds = self._initialize_parameters(data, K, random_state)
    else:
        weights = init_weights.copy()
        means = init_means.copy()
        stds = init_stds.copy()

    # EM цикл
    log_likelihood_prev = -np.inf

    for iteration in range(self.max_iter):
        # E-step
        responsibilities = self.e_step(data, weights, means, stds)

        # M-step
        weights, means, stds = self.m_step(data, responsibilities)

        # Обчислення log-likelihood
        log_likelihood = self.compute_log_likelihood(data, weights, means, stds)

        # Перевірка збіжності
        if abs(log_likelihood - log_likelihood_prev) < self.tolerance:
            if self.verbose:
                print(f"Збіжність досягнута на ітерації {iteration + 1}")
            break

        log_likelihood_prev = log_likelihood

    # Обчислення BIC та AIC
    n_params = K * 3 - 1  # (w, μ, σ) для кожного компонента, мінус 1 для обмеження на w
    bic = -2 * log_likelihood + n_params * np.log(n)
```

```
        aic = -2 * log_likelihood + 2 * n_params

        return GaussianMixtureParams(
            K=K,
            weights=weights,
            means=means,
            stds=stds,
            covariances=stds ** 2,
            bic_score=bic,
            aic_score=aic,
            log_likelihood=log_likelihood,
            n_iterations=iteration + 1
        )

    def _initialize_parameters(
        self,
        data: np.ndarray,
        K: int,
        random_state: Optional[int] = None
    ) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
        """
        Ініціалізація параметрів через k-means style підхід.

        Args:
            data: Дані
            K: Кількість компонентів
            random_state: Seed

        Returns:
            Tuple[np.ndarray, np.ndarray, np.ndarray]: weights, means, stds
        """
        rng = np.random.default_rng(random_state)

        # Рівномірні ваги
        weights = np.ones(K) / K

        # Середні: вибираємо випадкові точки з даних
        indices = rng.choice(len(data), size=K, replace=False)
        means = data[indices]

        # Стандартні відхилення: беремо загальне std
        stds = np.ones(K) * np.std(data)

        return weights, means, stds


class MixtureEstimator:
    """
    Оцінка параметрів суміші для даних про час обробки.

    Визначає оптимальну кількість компонентів K та оцінює параметри суміші.
    """

    def __init__(
        self,
        distribution_type: str = 'gaussian',
        max_components: int = 5,
```

```
    em_max_iter: int = 100,
    em_tolerance: float = 1e-6,
    verbose: bool = False
):
    """
    Ініціалізація MixtureEstimator.

    Args:
        distribution_type: Тип розподілу ('gaussian' або 'shifted_exponential')
        max_components: Максимальна кількість компонентів для пошуку
        em_max_iter: Максимальна кількість ітерацій EM
        em_tolerance: Толерантність для EM
        verbose: Виводити детальну інформацію
    """
    self.distribution_type = distribution_type
    self.max_components = max_components
    self.em_max_iter = em_max_iter
    self.em_tolerance = em_tolerance
    self.verbose = verbose

    self.em_algorithm = EMAlgorithm(
        max_iter=em_max_iter,
        tolerance=em_tolerance,
        verbose=verbose
    )

def estimate_optimal_K(
    self,
    data: np.ndarray,
    criterion: str = 'bic',
    random_state: Optional[int] = None
) -> int:
    """
    Оцінка оптимальної кількості компонентів K через BIC/AIC мінімізацію.

    Args:
        data: Дані для аналізу
        criterion: Критерій вибору ('bic' або 'aic')
        random_state: Seed

    Returns:
        int: Оптимальна кількість компонентів
    """
    if len(data) < self.max_components:
        warnings.warn(
            f"Недостатньо даних ({len(data)}) для max_components={self.max_components}. "
            f"Використовую K=1"
        )
        return 1

    scores = []
    K_values = range(1, min(self.max_components + 1, len(data) // 2))

    for K in K_values:
        try:
            params = self.em_algorithm.fit(data, K, random_state=random_state)
            score = params.bic_score if criterion == 'bic' else params.aic_score
```

```
            scores.append(score)

            if self.verbose:
                print(f"K={K}: {criterion.upper()}={score:.2f}")
        except Exception as e:
            if self.verbose:
                print(f"K={K}: Помилка - {str(e)}")
            scores.append(np.inf)

    # Вибір K з мінімальним BIC/AIC
    optimal_K = list(K_values)[np.argmin(scores)]

    if self.verbose:
        print(f"Оптимальна K: {optimal_K} ({criterion.upper()}={min(scores):.2f})")

    return optimal_K

def fit_gaussian_mixture(
    self,
    data: np.ndarray,
    K: Optional[int] = None,
    random_state: Optional[int] = None
) -> GaussianMixtureParams:
    """
    Оцінка параметрів гауссівської суміші.

    Args:
        data: Дані для fit
        K: Кількість компонентів (якщо None - автоматичний вибір)
        random_state: Seed

    Returns:
        GaussianMixtureParams: Оцінені параметри
    """
    data = np.atleast_1d(data).flatten()

    # Автоматичний вибір K якщо не вказано
    if K is None:
        K = self.estimate_optimal_K(data, criterion='bic', random_state=random_state)

    # Fit EM алгоритмом
    params = self.em_algorithm.fit(data, K, random_state=random_state)

    return params

def estimate_all_intervals(
    self,
    intervals: List[Tuple[float, float]],
    arrival_times: np.ndarray,
    processing_times: np.ndarray,
    random_state: Optional[int] = None
) -> List[GaussianMixtureParams]:
    """
    Оцінка параметрів суміші для кожного інтервалу.

    Args:
        intervals: Список інтервалів [(T_0, T_1), ...]
```

```
            arrival_times: Часи надходження задач
            processing_times: Часи обробки задач
            random_state: Seed

        Returns:
            List[GaussianMixtureParams]: Параметри для кожного інтервалу
        """
        all_params = []

        for idx, (t_start, t_end) in enumerate(intervals):
            # Фільтрація даних для інтервалу
            mask = (arrival_times >= t_start) & (arrival_times <= t_end)
            interval_data = processing_times[mask]

            if len(interval_data) == 0:
                warnings.warn(f"Інтервал {idx} порожній")
                continue

            if self.verbose:
                print(f"\nІнтервал {idx}: [{t_start:.2f}, {t_end:.2f}], n={len(interval_data)}")

            # Оцінка параметрів
            params = self.fit_gaussian_mixture(interval_data, K=None, random_state=random_state)
            all_params.append(params)

        return all_params

    def __repr__(self) -> str:
        """Зручне представлення MixtureEstimator."""
        return (
            f"MixtureEstimator(\n"
            f"  distribution='{self.distribution_type}',\n"
            f"  max_components={self.max_components},\n"
            f"  em_max_iter={self.em_max_iter}\n"
            f")"
        )

// Вміст файлу distribution_tester.py
"""
Статистичне тестування розподілів з використанням тесту Колмогорова-Смірнова.
"""
import numpy as np
from dataclasses import dataclass
from typing import List, Tuple, Optional
from scipy.stats import ks_2samp, kstest
import pandas as pd


@dataclass
class KSTestResult:
    """
    Результат тесту Колмогорова-Смірнова.

    Attributes:
        interval_pair: Пара індексів інтервалів (prev_idx, cur_idx)
        d_statistic: D статистика (sup|F_n1(x) - F_n2(x)|)
        p_value: P-value тесту
```

```
        is_different: Чи відхилено H_0 (True = розподіли різні)
        critical_value: Критичне значення для заданого α
        sample_sizes: Розміри вибірок (n1, n2)
        alpha: Рівень значущості
    """
    interval_pair: Tuple[int, int]
    d_statistic: float
    p_value: float
    is_different: bool
    critical_value: float
    sample_sizes: Tuple[int, int]
    alpha: float


class KolmogorovSmirnovTester:
    """
    Тест Колмогорова-Смірнова для порівняння розподілів між інтервалами.

    Перевіряє гіпотезу:
    H_0: F_prev(x) = F_cur(x)
    H_1: F_prev(x) ≠ F_cur(x)
    """

    def __init__(self, alpha: float = 0.05, verbose: bool = False):
        """
        Ініціалізація KS тестеру.

        Args:
            alpha: Рівень значущості (зазвичай 0.05)
            verbose: Виводити детальну інформацію

        Raises:
            ValueError: Якщо alpha не в (0, 1)
        """
        if not (0 < alpha < 1):
            raise ValueError("alpha має бути між 0 і 1")

        self.alpha = alpha
        self.verbose = verbose
        self.test_results = []

    def two_sample_ks_test(
        self,
        data1: np.ndarray,
        data2: np.ndarray,
        interval_pair: Optional[Tuple[int, int]] = None
    ) -> KSTestResult:
        """
        Двовибірковий тест Колмогорова-Смірнова.

        D = sup_x |F_n1(x) - F_n2(x)|

        Args:
            data1: Перша вибірка
            data2: Друга вибірка
            interval_pair: Пара індексів інтервалів (опційно)
```

```
    Returns:
        KSTestResult: Результат тесту
    """
    data1 = np.atleast_1d(data1).flatten()
    data2 = np.atleast_1d(data2).flatten()

    n1, n2 = len(data1), len(data2)

    if n1 == 0 or n2 == 0:
        raise ValueError("Вибірки не можуть бути порожніми")

    # Виконання KS тесту
    d_statistic, p_value = ks_2samp(data1, data2, alternative='two-sided')

    # Критичне значення для двовибіркового KS тесту
    critical_value = self._get_critical_value(n1, n2, self.alpha)

    # Прийняття рішення
    is_different = p_value < self.alpha

    result = KSTestResult(
        interval_pair=interval_pair if interval_pair else (0, 1),
        d_statistic=d_statistic,
        p_value=p_value,
        is_different=is_different,
        critical_value=critical_value,
        sample_sizes=(n1, n2),
        alpha=self.alpha
    )

    if self.verbose:
        self._print_test_result(result)

    return result

def compare_consecutive_intervals(
    self,
    all_intervals_data: List[np.ndarray]
) -> List[KSTestResult]:
    """
    Порівняння всіх послідовних пар інтервалів.

    Args:
        all_intervals_data: Список вибірок для кожного інтервалу

    Returns:
        List[KSTestResult]: Результати тестів для всіх пар
    """
    if len(all_intervals_data) < 2:
        raise ValueError("Потрібно хоча б 2 інтервали для порівняння")

    results = []

    for idx in range(len(all_intervals_data) - 1):
        data_prev = all_intervals_data[idx]
        data_cur = all_intervals_data[idx + 1]
```

```
            if self.verbose:
                print(f"\n{'='*60}")
                print(f"Порівняння інтервалів {idx} vs {idx + 1}")
                print(f"{'='*60}")

            result = self.two_sample_ks_test(
                data_prev,
                data_cur,
                interval_pair=(idx, idx + 1)
            )

            results.append(result)

        self.test_results = results
        return results

    def test_against_theoretical(
        self,
        data: np.ndarray,
        cdf_func,
        interval_idx: Optional[int] = None
    ) -> KSTestResult:
        """
        Тест на відповідність теоретичному розподілу.

        Args:
            data: Вибірка
            cdf_func: Теоретична функція розподілу (callable)
            interval_idx: Індекс інтервалу (опційно)

        Returns:
            KSTestResult: Результат тесту
        """
        data = np.atleast_1d(data).flatten()

        if len(data) == 0:
            raise ValueError("Вибірка не може бути порожньою")

        # Виконання KS тесту проти теоретичного розподілу
        d_statistic, p_value = kstest(data, cdf_func)

        # Критичне значення для одновибіркового тесту
        n = len(data)
        critical_value = self._get_critical_value_single_sample(n, self.alpha)

        is_different = p_value < self.alpha

        result = KSTestResult(
            interval_pair=(interval_idx, -1) if interval_idx is not None else (-1, -1),
            d_statistic=d_statistic,
            p_value=p_value,
            is_different=is_different,
            critical_value=critical_value,
            sample_sizes=(n, 0),
            alpha=self.alpha
        )
```

```
    if self.verbose:
        self._print_test_result(result, theoretical=True)

    return result

def _get_critical_value(self, n1: int, n2: int, alpha: float) -> float:
    """
    Критичне значення для двовибіркового KS тесту.

    Наближення:
    c(α) * sqrt[(n1 + n2) / (n1 * n2)]

    Args:
        n1: Розмір першої вибірки
        n2: Розмір другої вибірки
        alpha: Рівень значущості

    Returns:
        float: Критичне значення
    """
    # Таблиця критичних значень для різних α
    c_values = {
        0.10: 1.224,
        0.05: 1.358,
        0.025: 1.480,
        0.01: 1.628,
        0.005: 1.731,
    }

    # Вибір найближчого α
    c_alpha = c_values.get(alpha, 1.358)  # за замовчуванням 0.05

    critical_value = c_alpha * np.sqrt((n1 + n2) / (n1 * n2))

    return critical_value

def _get_critical_value_single_sample(self, n: int, alpha: float) -> float:
    """
    Критичне значення для одновибіркового KS тесту.

    Наближення: c(α) / sqrt(n)

    Args:
        n: Розмір вибірки
        alpha: Рівень значущості

    Returns:
        float: Критичне значення
    """
    c_values = {
        0.10: 1.224,
        0.05: 1.358,
        0.025: 1.480,
        0.01: 1.628,
        0.005: 1.731,
    }
```

```
    c_alpha = c_values.get(alpha, 1.358)
    critical_value = c_alpha / np.sqrt(n)

    return critical_value

def _print_test_result(self, result: KSTestResult, theoretical: bool = False):
    """
    Вивести результат тесту.

    Args:
        result: Результат тесту
        theoretical: Чи це тест проти теоретичного розподілу
    """
    if theoretical:
        print(f"Інтервал {result.interval_pair[0]} vs Теоретичний розподіл")
    else:
        print(f"Інтервали {result.interval_pair[0]} vs {result.interval_pair[1]}")

    print(f"  D статистика: {result.d_statistic:.6f}")
    print(f"  P-value: {result.p_value:.6f}")
    print(f"  Критичне значення (α={result.alpha}): {result.critical_value:.6f}")
    print(f"  Розміри вибірок: {result.sample_sizes}")

    if result.is_different:
        print(f"  Висновок: Розподіли РІЗНІ (H_0 відхилено)")
    else:
        print(f"  Висновок: Розподіли ОДНАКОВІ (H_0 не відхилено)")

def export_results_to_csv(self, output_path: str) -> None:
    """
    Експорт результатів тестів в CSV файл.

    Args:
        output_path: Шлях для збереження CSV
    """
    if not self.test_results:
        raise ValueError("Немає результатів для експорту")

    data = []
    for result in self.test_results:
        data.append({
            'interval_prev': result.interval_pair[0],
            'interval_cur': result.interval_pair[1],
            'd_statistic': result.d_statistic,
            'p_value': result.p_value,
            'critical_value': result.critical_value,
            'is_different': result.is_different,
            'n1': result.sample_sizes[0],
            'n2': result.sample_sizes[1],
            'alpha': result.alpha,
        })

    df = pd.DataFrame(data)
    df.to_csv(output_path, index=False)

    print(f"Результати KS тестів збережені: {output_path}")
```

```
    def get_summary_statistics(self) -> dict:
        """
        Отримати загальну статистику по всіх тестах.

        Returns:
            dict: Словник зі статистикою
        """
        if not self.test_results:
            return {}

        d_statistics = [r.d_statistic for r in self.test_results]
        p_values = [r.p_value for r in self.test_results]
        n_different = sum(1 for r in self.test_results if r.is_different)

        stats = {
            'n_tests': len(self.test_results),
            'n_different': n_different,
            'n_same': len(self.test_results) - n_different,
            'percent_different': 100 * n_different / len(self.test_results),
            'd_statistic': {
                'mean': np.mean(d_statistics),
                'std': np.std(d_statistics),
                'min': np.min(d_statistics),
                'max': np.max(d_statistics),
            },
            'p_value': {
                'mean': np.mean(p_values),
                'median': np.median(p_values),
                'min': np.min(p_values),
                'max': np.max(p_values),
            }
        }

        return stats

    def __repr__(self) -> str:
        """Зручне представлення KS тестеру."""
        return (
            f"KolmogorovSmirnovTester(\n"
            f"  alpha={self.alpha},\n"
            f"  n_tests={len(self.test_results)}\n"
            f")"
        )


// Вміст файлу monte_carlo_analyzer.py

"""
Monte Carlo аналіз результатів симуляції СМО.

Багаторазова симуляція для статистичного аналізу та порівняння систем.
"""
import numpy as np
from dataclasses import dataclass, field
from typing import List, Dict, Tuple, Optional, Callable
from collections import defaultdict
import json
```

```
# Імпорти локальних модулів
from .queueing_system_simulator import (
    SystemStats,
    QueueingSystemSimulator,
    PoissonProcessGenerator,
    ServerSelectionProtocol
)
from .mixture_estimator import GaussianMixtureParams


@dataclass
class SimulationConfig:
    """
    Конфігурація для однієї симуляції.

    Attributes:
        system_type: Тип системи ('single' або 'multi')
        protocol_type: Тип протоколу ('random' або 'optimal')
        M: Кількість потоків
        C: Кількість серверів
        mu_servers: Інтенсивності обробки серверів
        T: Час симуляції
        lambda_funcs: Функції інтенсивності для кожного потоку
        weights: Ваги для потоків
        mixture_params: Параметри суміші для генерації часів обслуговування
    """
    system_type: str  # 'single' або 'multi'
    protocol_type: str  # 'random' або 'optimal'
    M: int
    C: int
    mu_servers: List[float]
    T: float
    lambda_funcs: List[Callable[[float], float]]
    weights: List[float]
    mixture_params: GaussianMixtureParams


@dataclass
class DistributionStats:
    """
    Статистика розподілу метрики.

    Attributes:
        mean: Середнє значення
        std: Стандартне відхилення
        median: Медіана
        quantiles: Квантилі (0.05, 0.25, 0.75, 0.95)
        ci_lower: Нижня межа довірчого інтервалу
        ci_upper: Верхня межа довірчого інтервалу
        min: Мінімум
        max: Максимум
    """
    mean: float
    std: float
    median: float
    quantiles: Dict[float, float]
```

```
    ci_lower: float
    ci_upper: float
    min: float
    max: float


@dataclass
class ComparisonReport:
    """
    Звіт порівняння двох конфігурацій.

    Attributes:
        config1_name: Назва першої конфігурації
        config2_name: Назва другої конфігурації
        pk_comparison: Порівняння p_k розподілів
        error_comparison: Порівняння Error(T)
        queue_length_comparison: Порівняння середньої довжини черги
        waiting_time_comparison: Порівняння часу очікування
        improvement_percent: Покращення у відсотках
        statistical_significance: Чи є різниця статистично значущою
        p_value: P-value тесту
    """
    config1_name: str
    config2_name: str
    pk_comparison: Dict[int, Tuple[float, float]]
    error_comparison: Tuple[DistributionStats, DistributionStats]
    queue_length_comparison: Tuple[DistributionStats, DistributionStats]
    waiting_time_comparison: Tuple[DistributionStats, DistributionStats]
    improvement_percent: float
    statistical_significance: bool
    p_value: float


class MonteCarloAnalyzer:
    """
    Багаторазова симуляція для статистичного аналізу.
    """

    def __init__(
        self,
        num_runs: int = 100,
        confidence_level: float = 0.95,
        verbose: bool = False,
        random_state: Optional[int] = None
    ):
        """
        Ініціалізація Monte Carlo аналізатора.

        Args:
            num_runs: Кількість прогонів симуляції
            confidence_level: Рівень довіри для довірчих інтервалів
            verbose: Виводити детальну інформацію
            random_state: Seed для генератора
        """
        self.num_runs = num_runs
        self.confidence_level = confidence_level
        self.verbose = verbose
```

```
    self.random_state = random_state
    self.rng = np.random.default_rng(random_state)

def run_batch_simulations(
    self,
    config: SimulationConfig,
    simulator: QueueingSystemSimulator
) -> List[SystemStats]:
    """
    Запуск N симуляцій для однієї конфігурації.

    Args:
        config: Конфігурація симуляції
        simulator: Симулятор СМО

    Returns:
        List[SystemStats]: Результати всіх прогонів
    """
    all_results = []
    poisson_gen = PoissonProcessGenerator(random_state=self.random_state)

    for run_idx in range(self.num_runs):
        if self.verbose and run_idx % 10 == 0:
            print(f"Прогон {run_idx + 1}/{self.num_runs}")

        # Генерація надходжень
        if config.system_type == 'single':
            # Система 1: один потік зі змінною інтенсивністю
            lambda_combined = self._create_combined_lambda(
                config.lambda_funcs,
                config.weights
            )
            arrival_times = poisson_gen.generate_single_stream(
                lambda_combined,
                config.T
            )
            stream_ids = None
        else:
            # Система 2: багато потоків
            arrival_times, stream_ids = poisson_gen.generate_multi_stream(
                config.lambda_funcs,
                config.weights,
                config.T
            )

        # Генерація часів обслуговування
        service_times = config.mixture_params.sample(
            len(arrival_times),
            random_state=self.rng.integers(0, 2**31)
        )
        service_times = np.maximum(service_times, 1e-6)

        # Симуляція
        if config.system_type == 'single':
            stats = simulator.simulate_system1(arrival_times, service_times, config.T)
        else:
            stats = simulator.simulate_system2(
```

```
                arrival_times,
                service_times,
                stream_ids,
                config.T
            )

        all_results.append(stats)

    return all_results

def compare_systems(
    self,
    results1: List[SystemStats],
    results2: List[SystemStats],
    config1_name: str = "System 1",
    config2_name: str = "System 2"
) -> ComparisonReport:
    """
    Порівняння двох систем.

    Args:
        results1: Результати першої системи
        results2: Результати другої системи
        config1_name: Назва першої системи
        config2_name: Назва другої системи

    Returns:
        ComparisonReport: Звіт порівняння
    """
    # Агрегація метрик
    error_stats1 = self._aggregate_error_rates(results1)
    error_stats2 = self._aggregate_error_rates(results2)

    queue_stats1 = self._aggregate_queue_lengths(results1)
    queue_stats2 = self._aggregate_queue_lengths(results2)

    waiting_stats1 = self._aggregate_waiting_times(results1)
    waiting_stats2 = self._aggregate_waiting_times(results2)

    pk_comp = self._compare_pk_distributions(results1, results2)

    # Статистична значущість (t-test для error rates)
    error_values1 = [r.error_rate for r in results1]
    error_values2 = [r.error_rate for r in results2]
    p_value = self._t_test(error_values1, error_values2)

    # Покращення
    improvement = 0.0
    if error_stats1.mean > 0:
        improvement = (error_stats1.mean - error_stats2.mean) / error_stats1.mean * 100

    return ComparisonReport(
        config1_name=config1_name,
        config2_name=config2_name,
        pk_comparison=pk_comp,
        error_comparison=(error_stats1, error_stats2),
        queue_length_comparison=(queue_stats1, queue_stats2),
```

```
            waiting_time_comparison=(waiting_stats1, waiting_stats2),
            improvement_percent=improvement,
            statistical_significance=(p_value < 0.05),
            p_value=p_value
        )

    def compare_protocols(
        self,
        random_results: List[SystemStats],
        optimal_results: List[SystemStats]
    ) -> ComparisonReport:
        """
        Порівняння двох протоколів (Random vs Optimal).

        Args:
            random_results: Результати з випадковим протоколом
            optimal_results: Результати з оптимальним протоколом

        Returns:
            ComparisonReport: Звіт порівняння
        """
        return self.compare_systems(
            random_results,
            optimal_results,
            config1_name="Random Protocol",
            config2_name="Optimal Protocol"
        )

    def aggregate_pk_distributions(
        self,
        all_results: List[SystemStats]
    ) -> Dict[int, DistributionStats]:
        """
        Агрегація p_k розподілів з усіх прогонів.

        Args:
            all_results: Результати всіх прогонів

        Returns:
            Dict[int, DistributionStats]: Статистика для кожного k
        """
        # Збір всіх p_k для кожного k
        pk_values = defaultdict(list)

        for result in all_results:
            for k, prob in result.pk_distribution.items():
                pk_values[k].append(prob)

        # Обчислення статистики для кожного k
        pk_stats = {}
        for k, values in pk_values.items():
            pk_stats[k] = self._compute_distribution_stats(np.array(values))

        return pk_stats

    def _aggregate_error_rates(
        self,
```

```
    results: List[SystemStats]
) -> DistributionStats:
    """Агрегація Error(T)."""
    error_rates = np.array([r.error_rate for r in results])
    return self._compute_distribution_stats(error_rates)

def _aggregate_queue_lengths(
    self,
    results: List[SystemStats]
) -> DistributionStats:
    """Агрегація середніх довжин черги."""
    queue_lengths = np.array([r.avg_queue_length for r in results])
    return self._compute_distribution_stats(queue_lengths)

def _aggregate_waiting_times(
    self,
    results: List[SystemStats]
) -> DistributionStats:
    """Агрегація часів очікування."""
    waiting_times = np.array([r.avg_waiting_time for r in results])
    return self._compute_distribution_stats(waiting_times)

def _compute_distribution_stats(
    self,
    data: np.ndarray
) -> DistributionStats:
    """
    Обчислення статистики розподілу.

    Args:
        data: Дані

    Returns:
        DistributionStats: Статистика
    """
    if len(data) == 0:
        return DistributionStats(
            mean=0.0, std=0.0, median=0.0,
            quantiles={}, ci_lower=0.0, ci_upper=0.0,
            min=0.0, max=0.0
        )

    mean = float(np.mean(data))
    std = float(np.std(data, ddof=1)) if len(data) > 1 else 0.0
    median = float(np.median(data))

    quantiles = {
        0.05: float(np.percentile(data, 5)),
        0.25: float(np.percentile(data, 25)),
        0.75: float(np.percentile(data, 75)),
        0.95: float(np.percentile(data, 95)),
    }

    # Довірчі інтервали
    ci_lower, ci_upper = self.compute_confidence_intervals(data)

    return DistributionStats(
```

```
            mean=mean,
            std=std,
            median=median,
            quantiles=quantiles,
            ci_lower=ci_lower,
            ci_upper=ci_upper,
            min=float(np.min(data)),
            max=float(np.max(data))
        )

    def compute_confidence_intervals(
        self,
        data: np.ndarray
    ) -> Tuple[float, float]:
        """
        Обчислення довірчих інтервалів.

        Args:
            data: Дані

        Returns:
            Tuple[float, float]: (нижня межа, верхня межа)
        """
        if len(data) == 0:
            return 0.0, 0.0

        alpha = 1 - self.confidence_level
        lower_percentile = (alpha / 2) * 100
        upper_percentile = (1 - alpha / 2) * 100

        ci_lower = float(np.percentile(data, lower_percentile))
        ci_upper = float(np.percentile(data, upper_percentile))

        return ci_lower, ci_upper

    def _compare_pk_distributions(
        self,
        results1: List[SystemStats],
        results2: List[SystemStats]
    ) -> Dict[int, Tuple[float, float]]:
        """
        Порівняння p_k розподілів між двома системами.

        Returns:
            Dict[int, Tuple[float, float]]: {k: (mean1, mean2)}
        """
        pk_stats1 = self.aggregate_pk_distributions(results1)
        pk_stats2 = self.aggregate_pk_distributions(results2)

        # Об'єднуємо всі k
        all_k = set(pk_stats1.keys()) | set(pk_stats2.keys())

        comparison = {}
        for k in sorted(all_k):
            mean1 = pk_stats1.get(k, DistributionStats(0, 0, 0, {}, 0, 0, 0, 0)).mean
            mean2 = pk_stats2.get(k, DistributionStats(0, 0, 0, {}, 0, 0, 0, 0)).mean
            comparison[k] = (mean1, mean2)
```

```
    return comparison

def _t_test(
    self,
    data1: List[float],
    data2: List[float]
) -> float:
    """
    Двовибірковий t-test.

    Args:
        data1: Перша вибірка
        data2: Друга вибірка

    Returns:
        float: P-value
    """
    from scipy.stats import ttest_ind

    if len(data1) < 2 or len(data2) < 2:
        return 1.0

    _, p_value = ttest_ind(data1, data2)
    return float(p_value)

def _create_combined_lambda(
    self,
    lambda_funcs: List[Callable[[float], float]],
    weights: List[float]
) -> Callable[[float], float]:
    """
    Створення комбінованої функції інтенсивності.

    λ(t) = Σ w_i * λ_i(t)

    Args:
        lambda_funcs: Функції інтенсивності
        weights: Ваги

    Returns:
        Callable: Комбінована функція
    """
    def combined_lambda(t: float) -> float:
        total = sum(w * lam(t) for w, lam in zip(weights, lambda_funcs))
        return total

    return combined_lambda

def export_results_to_json(
    self,
    all_results: List[SystemStats],
    output_path: str
) -> None:
    """
    Експорт результатів у JSON.
```

```
    Args:
        all_results: Результати симуляцій
        output_path: Шлях для збереження
    """
    data = []
    for idx, result in enumerate(all_results):
        data.append({
            'run_id': idx,
            'total_arrivals': result.total_arrivals,
            'total_departures': result.total_departures,
            'total_rejections': result.total_rejections,
            'error_rate': result.error_rate,
            'avg_queue_length': result.avg_queue_length,
            'avg_waiting_time': result.avg_waiting_time,
            'max_queue_length': result.max_queue_length,
            'pk_distribution': result.pk_distribution,
        })

    with open(output_path, 'w', encoding='utf-8') as f:
        json.dump(data, f, indent=2, ensure_ascii=False)

    print(f"Результати Monte Carlo збережено: {output_path}")

def __repr__(self) -> str:
    """Зручне представлення аналізатора."""
    return (
        f"MonteCarloAnalyzer(\n"
        f"  num_runs={self.num_runs},\n"
        f"  confidence_level={self.confidence_level}\n"
        f")"
    )
```