ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ імені І.І.МЕЧНИКОВА
(повне найменування вищого навчального закладу)

Факультет математики, фізики та інформаційних технологій
(повне найменування інституту, назва факультету (відділення))

Кафедра математичного забезпечення комп'ютерних систем
(повна назва кафедри (предметної, циклової комісії))

Кваліфікаційна робота

на здобуття рівня вищої освіти «магістр»
(рівень вищої освіти)

Методи та інформаційні технології масштабування web додатків

Techniques and information technologies for web application scalability

Виконав: студент заочної форми навчання
спеціальності 126 – Інформаційні системи та технології.
(шифр і назва напряму підготовки, спеціальності)
Освітня програма «Інформаційні системи та технології»

(назва освітньої програми)
Камєнєв Кирило Ігорович
(прізвище, ім'я, по-батькові)

Керівник д.т.н., проф. Малахов Є.В.
(науковий ступінь, вчене звання, прізвище та ініціали, **підпис**)

Рецензент ____________________
(науковий ступінь, вчене звання, прізвище та ініціали)

Рецензент ____________________
(науковий ступінь, вчене звання, прізвище та ініціали)

Рекомендовано до захисту:
Протокол засідання кафедри
№___ від «___»_____ 2024 р.
Завідувач кафедри
_____________Євгеній МАЛАХОВ_
(підпис) (ім'я, прізвище)

Захищено на засіданні ЕК № ___
протокол № __ від «___» ____ 2024 р.
Оцінка / /_____
(за національною шкалою, шкалою ECTS, бали)
Голова ЕК
_______ _Володимир ВИЧУЖАНІН_
(підпис) (ім'я, прізвище)

Одеса – 2024

АНОТАЦІЯ

У цьому дослідженні розглядаються сучасні методи та технології масштабування веб-додатків, що дозволяють вирішувати проблеми, пов'язані зі зростаючим попитом та мінливим трафіком. У дослідженні проаналізовано підходи до вертикального та горизонтального масштабування, використання розподілених баз даних, мікросервісів та безсерверних архітектур, з акцентом на хмарні платформи, такі як AWS Lambda та Amazon Aurora. Особливу увагу приділено практичному застосуванню та оцінці безсерверних рішень для досягнення ефективної масштабованості. Висновки надають структуровану оцінку стратегій масштабування, акцентуючи увагу на архітектурних компромісах та їхньому впливі на продуктивність і оптимізацію ресурсів у веб-додатках.

ABSTRACT

This study examines modern techniques and technologies for scaling web applications, addressing the challenges posed by growing user demand and fluctuating traffic. The research analyzes vertical and horizontal scaling approaches, the use of distributed databases, microservices, and serverless architectures, focusing on cloud platforms such as AWS Lambda and Amazon Aurora. Special attention is given to the practical application and evaluation of serverless solutions for achieving efficient scalability. The findings provide a structured assessment of scaling strategies, emphasizing architectural trade-offs and their impact on performance and resource optimization in web applications.

ЗМІСТ

ПЕРЕЛІК ПРИЙНЯТИХ СКОРОЧЕНЬ... 5

ВСТУП 6

РОЗДІЛ 1. АНАЛІЗ ЛІТЕРАТУРНИХ ДЖЕРЕЛ З ПРОБЛЕМИ МАСШТАБУВАННЯ WEB ДОДАТКІВ 10

1.1 Основні напрямки досліджень проблеми масштабування web додатків 10

1.2 Висновки за розділом 20

РОЗДІЛ 2. СЕРВІСИ БЕЗСЕРВЕРНИХ ОБЧИСЛЕНЬ ТА КЕРОВАНИХ БАЗ ДАНИХ 21

2.1 Amazon Web Services 22

2.1.1 AWS Lambda.. 24

2.1.2 Amazon Aurora.. 29

2.2 Microsoft Azure 35

2.2.1 Azure Functions... 37

2.2.2 Azure SQL Database 41

2.3 Google Cloud 47

2.3.1 Cloud Run functions 49

2.3.2 Spanner 54

2.4 Висновки по розділу 59

РОЗДІЛ 3. ОЦІНКА ПРИДАТНОСТІ БЕЗСЕРВЕРНИХ РІШЕНЬ ДЛЯ МАСШТАБОВАНИХ СЕРВІСІВ НА ПРИКЛАДІ AWS................................. 64

3.1 Аналіз впливу виділених потужностей Lambda на час виконання функції 65

3.2 Аналіз впливу масштабування Lambda та Aurora на час виконання функції 73
3.3 Висновки по розділу 77
ВИСНОВКИ 78
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ 79
ДОДАТОК А Код функції Lambda 87
ДОДАТОК Б Графіки залежності кількості запитів від ACU 93
ДОДАТОК В Графіки залежності часу обробки запитів від ACU 97

ПЕРЕЛІК ПРИЙНЯТИХ СКОРОЧЕНЬ

API – Application Programming Interface / програмний інтерфейс додатку

AWS – Amazon Web Services

AZ – Availability Zones / зони доступності

DDL – Data Definition Language / мова визначення даних

DTU – Database transaction units – одиниці транзакції БД

FAAS – Function as a Service / функція як сервіс

HDFS – Hadoop Distributed File System

IOT – Internet of things / інтернет речей

LSMT – Log -structured merge-tree / LSM дерево

OCC – Optimistic Concurrency Control / оптимістичний контроль паралелізму

OLAP – Online Analytical Processing / аналітична обробка у реальному часі

OLTP – Online Transaction Processing / онлайнова обробка транзакцій

TLS – Transport Layer Security / безпека на транспортному рівні

TTFB – Time to first byte / час до першого байту

vCPU – Virtual CPU / віртуальний центральний процесор

VPC – Virtual Private Cloud / приватна віртуальна хмара

ВСТУП

Згідно з даними на кінець 2021 року понад 5 мільярдів людей користуються інтернетом, що складає зріст у 1392% з 2000 року [1]. Глобальні онлайн платформи зберігають і оброблюють величезні обсяги даних. Хоча більшість компаній не публікують точні дані, деяку інформацію можна знайти у їх блогах, наприклад, кількість посланих твітів (tweets) вимірюється у сотнях мільйонів щоденно [2], Instagram має більш ніж 2 мільярди активних користувачів щомісячно [3], Uber – 131 мільйон користувачів щомісячно [4], тощо. Окрім зросту кількості користувачів, такі платформи також мають опрацьовувати підвищені обсяги трафіку протягом певних годин, наприклад, Facebook має пікове навантаження ввечері за європейским часом [5]. Операція таких систем вимагає приділення особливої уваги до їх *масштабованості*.

Масштабовність (scalability) визначає здатність системи вміщувати постійно зростаючу кількість елементів або об’єктів, безпроблемно опрацьовувати зростаючи обсяги роботи, та/або бути сприйнятливою до зросту [6].

У контексті програмного забезпечення, зріст може визначатися за різними параметрами, такими як:

- Кількість одночасних запитів користувачів (або зовнішніх) які можуть бути оброблені системою.
- Обсяг даних, який може бути ефективно опрацьований.
- Вартість, яку можна бути отримати з даних, що знаходяться у системі, за допомогою прогнозної аналітики.
- Можливість підтримувати стабільний, постійний час відгуку при зрості навантаження запитів [7].

Якщо при проектуванні системи її масштабованість не була передбачена заздалегідь, це може привести до сотен мільйонів [8], або навіть мільйярдів [9] доларів втрат.

Масштабовність досягається за допомогою двох підходів, а саме збільшення ємності (capacity) системи та оптимізація її компонентів. У свою чергу, збільшення ємності системи може бути досягнуто також двома методами: *вертикальне масштабування* (vertical scaling / scaling up) та *горизонтальне масштабування* (horizontal scaling / scaling out). При цьому, найчастіше не менш важливою є можливість динамічно зменшувати ємність системи для зменшення витрат.

Вертикальне масштабування позначає додавання ресурсів до окремого вузла системи, що зазвичай залучає додавання процесорів або пам’яті до окремого комп’ютера. Таке вертикальне масштабування існуючих систем також дозволяє більш ефективне використання технології віртуалізації.

Горизонтальне масштабування позначає додавання вузлів до системи, наприклад, додаткових комп’ютерів до розподіленого (distributed) додатку [10].

Горизонтальне масштабування покладається на можливість реплікації сервісів та розміщення цих сервісів на різних вузлах. Як правило, воно вимагає наявності двох елементів, а саме: *балансиру навантажень* (load balancer) та *безстанових сервісів* (stateless services).

Балансир навантажень – компонент системи, який перенаправляє запити до відповідних сервісів та повертає результат до клієнтів. Існує декілька стратегій, за якими обираються ці сервіси, проте головною ціллю є рівномірний розподіл навантаження.

Безстанові сервіси – такі сервіси, які не зберігають даних щодо окремих клієнтських сесій. Зазвичай дані щодо сесій зберігаються окремо таким чином, що будь який сервіс може отримати їх окремо.

Горизонтальне масштабування теоретично дозволяє додавати сервіси для обробки зростаючого обсягу даних, уникаючи зросту часу відгуку (response time). Наприклад, Amazon.com використовує понад 100 сервісів для

повернення одної сторінки користувачу [11]. При цьому, якщо один з таких сервісів перестає працювати, інші можуть перейняти навантаження, що підвищує стійкість системи. Проте, у цьому випадку, наступний шар стає лімітуючим, а саме – база даних (БД).

Вертикальне масштабування БД може бути ефективним для додатків з обмеженою кількістю користувачів. Наприклад, GCP (Google Cloud Platform) пропонує n1-highmem-96 варіант з 96 віртуальними процесорами (vCPU) та 624 GB RAM [12]. Окрім цього одним з підходів є зменшення обсягу запитів до БД. Це можна досягти за допомогою використання *розподіленого кешу* (distributed cache). Кеш зберігає часто запитувані дані у пам'яті, таким чином вони можуть бути швидко відправлені без навантаження БД. Тоді при запиті від зовнішніх сервісів спочатку перевіряється кеш, і тільки у випадку відсутності відповідних даних у ньому робиться запит до БД. Однак багато систем вимагають доступу до дуже великих обсягів даних, що робить використання одної БД практично неможливим. Додатково, згідно з законом Амдала [13], ефективність додавання процесорів до окремого вузла сильно залежить від того, яка частина програми, у процентах, виконується паралельно. Наприклад, якщо 50% програми виконується послідовно, то додавання більше 8 ядер не надає майже ніякого ефекту. У таких випадках існує два варіанти вирішення цієї проблеми: оптимізація коду та використання розподілених БД.

Розподілені БД можна поділити на дві основні категорії:

- Розподілені *SQL* (structured query language) БД, які представляють собою єдину логічну БД, яка розподілена по декількох фізичних вузлах. До них також відносяться NewSQL БД.
- Розподілені *NoSQL* (not only SQL) БД, що використовують різноманітні моделі та мови запитів та зберігають дані на окремих вузлах, доступ до яких можливий окремо.

Розподілені БД також покращують доступність (availability) системи. Вони підтримують реплікацію вузлів, які зберігають дані, що дозволяє доступ до даних у випадку проблем доступу до початкового вузла.

Хоча розподілені БД мають багато переваг, вони, як і інші компоненти масштабованих систем, вимагають компромісів. Для баз даних головним компромісом є узгодженість (consistency).

Вищезазначені компоненти можуть бути згруповані у різні шари, кожний зі своїм балансиром навантажень, кешами та сервісами, які, у свою чергу, пов'язані з іншими шарами, і т.д. Багато сучасних сервісів використовують мікросервіси, які, як правило, є більш незалежними та спеціалізованими.

У цілому, можливість додатку швидко та ефективно масштабуватися може бути його визначальною ознакою, що обумовлює його успіх або провал, вимагає ретельного попереднього планування його архітектури, та має низку компромісів, яким необхідно приділяти особливу увагу.

Метою роботи є підвищення ефективності масштабування web додатків.

Об'єктом дослідження є процес масштабування web додатків. Предметом дослідження є методи і підходи до масштабування web додатків.

РОЗДІЛ 1. АНАЛІЗ ЛІТЕРАТУРНИХ ДЖЕРЕЛ З ПРОБЛЕМИ МАСШТАБУВАННЯ WEB ДОДАТКІВ

1.1 Основні напрямки досліджень проблеми масштабування web додатків

Масштабування web додатків є багатошаровим комплексним питанням, яке розглядається з багатьох напрямків, таких як оптимізація клієнтів, проміжних вузлів, БД, серверів тощо.

Збільшення бази користувачів і запропонованих сервісів веб-додатків сприяє стрімкому зростанню різноманітності та складності серверних служб, з яких вони складаються [14]. Наразі спостерігається тенденція використання мікросервісних моделей [15] [16] [17], у яких програма розкладається на окремі мікросервіси [18], кожен з яких пропонує спеціалізовані функції, такі як управління HTTP (Hypertext Transfer Protocol) підключенням, маршрутизація протоколів [19], подача реклами [20], тощо. Такі моделі забезпечують незалежну масштабованість компонентів за допомогою адаптації кількості фізичних серверів/потужностей, призначених для кожного мікросервісу, у залежності від щоденних та довгострокових тенденцій навантаження [17].

У [21] авторами описуються модифікації до Apache Hadoop для використання у системі з високою пропускною здатністю (Facebook messages). Вони включають зміни до HDFS (Hadoop Distributed File System / розподілена файлова система Hadoop), такі як:

- Використання двох AvatarNode – активного і пасивного (standby) замість одного головного (master) NameNode. Обидва AvatarNodes отримують дані від DataNodes, що дозволяє пасивному AvatarNode підтримувати актуальний стан та дає йому можливість стати активним за менш ніж хвилину, на відміну від вбудованого BackupNode, для якого цей процес займає до 20

хвилин. Додатково, замість запису до логу транзакцій id блоків тільки при закритті файлу, виконується запис при виділенні кожного блоку, що дозволяє клієнтам продовжувати запис файлів у випадку аварійної заміни активного AvatarNode.

- Розробка DistributedAvatarFileSystem (DAFS), яка забезпечує доступ до HDFS та робить процес аварійної заміни гловного вузла прозорим для клієнтів, покращено Hadoop RPC (Remote Procedure Call) ПЗ, яке автоматично визначає версію ПЗ на сервері та використовує відповідний протокол.
- Модифікація методу розміщення блоків по серверам та стелажам, яка дозволяє зменшити вірогідність втрати даних у 100 раз, у порівнянні з початковим, випадковим методом.
- Покращення для обробки даних у реальному часі, які включають використання таймауту у Hadoop-RPCs, зчитування даних з локальних реплік та API (Application Programming Interface / програмний інтерфейс додатку) для прискорення доступу до файлу, запис до якого був завершений невірно.

Також описуються нові функції, а саме HDFS sync, що дозволяє запис коли Hflush/sync операція очікує відповіді, та можливість одночасного запису та читання з файлу через перерахунок контрольної суми останнього блоку даних.

Автори [10] розглядають питання масштабованості, еластичності і автономності БД у хмарі, використовуючи два підходи. Перший – злиття даних (Data Fusion), який передбачає використання сховищ ключ-значення (key-value stores), та подальше групування елементів, що дозволяє забезпечити багатоключовий доступ (multi key access) до даних. Прикладом такої системи є MegaStore від Google. Другий підхід – поділ даних (Data Fission), передбачає розділення великої БД на осколки (shards). До таких систем відносяться MS SQL Azure, ElasTraS, та ін. У статті представляються G-Store – масштабоване сховище даних, яке пропонує багатоключовий доступ за динамічними, не

перетинаючимися групами ключей; Iterative Copy – техніка для міграції вживу (live database migration) БД, які використовують архітектуру спільного сховища (shared storage architecture); Zephyr – техніка для міграції БД, які використовують архітектуру з розподільними сховищами (shared nothing architecture).

У [22] описується використання HBase та модифікації до неї для використання у Facebook Messages. Модифікації включають:

- підтримку гарантованих мультиколонкових транзакцій
- можливість оновлення ПЗ кластерів без призупинення БД цілком,
- оптимізації пошуку,
- формат файлів БД – HFile V2, що підтримує багаторівневі індекси,
- оптимізацію ущільнення (compaction) – процесу об'єднання файлів БД, який став враховувати не тільки файли, які підлягають ушільненню зараз, але і файли, які створюються в результаті цього процесу,
- підтримку розподіленого поділу логів по різним серверам, та інші.

Варто зазначити, що через декілька років ця система була мігрована до MyRocks – MySQL підсистеми зберігання (storage engine), оптимізованої щодо розміру даних та швидкості запису [23] [24]. Ця міграція дозволила спростити схему БД, зменшити фактор реплікації даних з 6 до 3, використовувати алгоритм компресії Zstandard [25], та у цілому зменшити розмір даних на 90%. Інші переваги нової системи включають спрощення відновлення сервісу у випадку аварії, оптимізація використання флеш-пам'яті, яка призвела до зменшення латентності читання (read latency) у 50 разів, та ін.

MyRocks використовує RocksDB [26] [27] – вбудоване (embedded) сховище ключ-значення. RocksDB представляє собою C++ бібліотеку, яка зберігає ключі у відсортированих послідовностях. Ключі та значення є довільними потоками байтів; нові записи розміщуються у вільних місцях, а фоновий процес ущільнення усуває дублікати та оброблює мітки видалення. БД не використовує фіксовані розміри сторінок; дані зберігаються у вигляді

LSM (Log -structured merge-tree) дерева [28], що дозволяє знизити кількість випадкових читань та записів. RocksDB підтримує атомарний запис множини ключів у БД, та ітерацію за ними. БД поділена на дві частини: тільки читання та читання/запис, що дозволяє знизити використання блокування (locks), окрім цього підтримується MVCC (Multiversion concurrency control – керування паралельним доступом за допомогою багатоверсійності) для транзакцій читання. Архітектура сховища дозволяє легко змінювати частини системи, наприклад, підтримуються різноманітні модулі стиснення (snappy, zlib, bzip, і т.д.), які можна підключати без впровадження змін до решти системи. У цілому ця БД найкраще підходить для додатків, які потребують низьку затримку при доступі до БД, таких як, додаток для виявлення спаму, черга повідомлень з великою кількістю вставок та видалень, тощо.

Стаття [29] представляє порівняльний аналіз продуктивності різних підсистем зберігання, а саме Apache Cassandra, Apache HBase, Project Voldemort, Redis, VoltDB, та MySQL InnoDB. Експерименти було поставлено у двох апаратних середовищах: обмеженим дисковими параметрами (disk-bound), тобто такому, де розмір даних не уміщався у оперативну пам’ять та вимушував використовувати жорсткі диски комп’ютерів, та обмеженим оперативною пам’яттю. Аналіз, який складався з різноманітних комбінацій операцій читання, запису та сканування, було проведено за допомогою використання Yahoo! Cloud Serving Benchmark. Результати показали лінійну масштабованість Cassandra, HBase та Project Voldemort у більшості тетсів, пропускна здатність Cassandra була найкращою, проте її латентність (latency) була незвичайно великою. Project Voldemort мав нижчу стабільну латентність. HBase мав найнижчу пропускну здатність з цих трьох систем, проте показав низьку латентність запису за рахунок високої латентності читання. Осколкові системи (sharded: Redis & MySQL) мали високу пропускну здатність, яка гірше масштабувалася. Ну і нарешті VoltDB показав високу пропускну здатність одного вузла, яка не масштабувалася до багатьох.

У [30] порівнюються Apache Cassandra та Apache HBase на хмарних сервісах Amazon EC2. Авторами було повторено та розширено ексерименти з [29] щодо швидкості роботи та еластичності цих систем. Результати показують, що Cassandra та HBase масштабуються майже лінійно, при цьому швидкість операцій читання Cassandra вище ніж HBase, а швидкість запису – навпаки. Т.я. результати тестування Cassandra показали найбільший розбіг з початковою роботою, було проведено додатковий аналіз цієї системи на різних конфігураціях EC2. Автори зазначають, що продуктивність системи сильно залежить від конфігурації, а також типу та обсягу збережених даних. Також висвітлюється компроміс між швидкостю масштабування та мінливістю (variability) продуктивності системи.

Автори [31] представляють розробку та оцінку DBX – резидентної БД, яка використовує обмежену транзакційну пам’ять Intel (RTM – Restricted Transactional Memory). RTM є частиною Intel TSX (Transactional Synchronization Extensions / Розширень Синхронізації Транзакцій), яке додає нові інструкції до існуючого набору інструкцій ЦП. Ці додаткові інструкції використовуються для початку, завершення та керування апаратними транзакціями. RTM має обмеження на розмір робочої множини, т.я. вона реалізується через кеш ЦП для відстеження записів/читань. Для боротьби з цією лімітацією, DBX складається з двох модулів: сховища ключ-значення та транзакційного шару. Сховище підтримує впорядкований доступ, використовуючи B+ дерево, та невпорядкований, використовуючи хеш-таблицю. Транзакційний шар впроваджує OCC (Optimistic Concurrency Control / Опитмістичний Контроль Паралельності), який припускає, що транзакції найчастіше виконуються, не заважаючи одна одній та дозволяє відокремити виконання транзакції до її підтвердження (commit); тільки стадія підтвердження використовує RTM для синхронізації. Ці модулі використовують окремі RTM області для синхронізації доступу до пам’яті.

Авторам вдалося досягти 506817 транзакцій за секунду на 8поточному Haswell ЦП.

У статті [17] порівнюється монолітна та мікросервісна архітектури у хмарних WEB додатках. Автори розробили та розгорнули (deployed) додаток використовуючи Play web framework. У роботі зазначається, що мікросервіси мають багато проблем, притаманних розподіленим системам, таких як таймаути, федерація даних, розподілення обов'язків, розподілені транзакціі, тощо. Більшість з них не виникає у монолітних додатках. Окрім цього розгорнення кожного мікросервісу вимагає спеціальної кофігурації серверів; при розгоренні нових версій дуже легко «зламати» зовнішні залежні сервіси. До переваг віднесено можливість масштабування кожного сервісу окремо Для оцінки продуктивності було використано JMeter від Amazon Web Services (AWS). У результаті аналізу було визначено, що використання мікросервісів не викликає значних змін латентності не зважаючи на більшу кількість хостів (hosts), проте дає більш гранульований контроль над обраними сервісами, що дозволило знизити витрати на інфраструктуру на 17%. У роботі зазначається, що для додатків з невеликою кількістю користувачів (сотні або тисячі) використання монолітного підходу може бути більш практичним.

Робота [32] пропонує альтернативну архітектуру для розподілених БД. Вона базується на двох принципах, а саме відділення обробки запитів від сховища даних, та спільне використання даних вузлами (sharing data), що оброблюють запити. Дані зберігаються у розподіленому менеджері записів, який складається з множини вузлів-сховищ. Шар обробки складається з множини вузлів обробки, які мають доступ до спільної системи сховища (storage system). Пропонується механізм вилучення даних безпосередньо з вузла, у якому вони знаходяться. Вузли обробки та сховища можуть масштабуватися окремо, у залежності від конкретних вимог. Архітектура підтримує як OLTP (Online Transaction Processing / онлайнова обробка транзакцій), так і OLAP (Online Analytical Processing / аналітична обробка у

реальному часі) навантаження. Автори зазначають, що запропонована архітектура найкраще підходить для локальних мереж (LAN – local area network), т.я. вона покладається на низьку латентність, інтенсивний обмін даними, обмежену буферізацію та синхронну реплікацію для запобігання втрати даних у випадку аварії. Використовуєтья TPC-C бенчмарк для аналізу та порівняння системи з VoltDB, MySQL Cluster та FoundationDB; результати показують конкурентоспроможність представленої системи.

Велика кількість публікацій присвячена балансирам навантажень. Так, роботи [33], [34], [35], [36], [37], [38] пропонують огляди, класифікації та порівняльні аналізи найбільш розповсюджених алгоритмів у хмарних середовищах. Балансири порівнюються за рядом показників, таких як пропускна здібність, час відгуку, масштабованість, час міграції між вузлами, відмовостійкість, споживання енергії, тощо. Пропонується ряд підходів до балансування навантажень, таких як використання еволюційних алгоритмів (генетичний, мурашиний, тощо) [33], мультиагентних систем [39], вірогіднисних розподілень [40] та ін.

У статті [41] розглядається питання динамічної вертикальної еластичності додатків на хмарних сервісах. Автори пропонують багатоступінчатий процес, у якому спочатку застосовується алгоритм машинного навчання, заснований на Гауссівському процесі, який використовується для прогнозування динамічного навантаження. На наступному кроці застосовується кластерізація методом k-середніх різних показників взаємодії з іншими робочими навантаженнями серверу. Після цього застосовується ще одна Гаусівська модель для вивчення продуктивності додатку, використовуючи виміряні дані, що надає прогнозний аналіз продуктивності додатку у реальному часі. Авторами зазначається до 39.46% менша хвостова латентність (tail latency) у порівнянні з реактивними підходами.

Авторами [42] розглядається горизонтальне та вертикальне масштабування додатків у контейнерах, використовуючи навчання з підкріпленням (Reinforcement Learning). Пропонуються методи, що базуються на Q-навчанні (Q-learning), Dyna-Q навчанні (Dyna-Q learning) та навчанні з підкріпленням, засноване на моделі (Model-Based Reinforcement Learning) для ситуацій з різними ступенями знайомства з системою. Запропоновані методи виконують як вертикальне, так і горизонтальне масштабування. У результаті експериментів проведених за допомогою розробленого авторами розширення до Docker [43], найкращі результати показує навчання з підкріпленням, засноване на моделі.

Робота [44] присвячена розробці методу оцінки ліміту навантаження на мікросервіси при дотриманні цілі рівню послуг (Service Level Objective / SLO); такий ліміт автори називають ємністю мікросервісу. Пропонується розміщення мікросервісів по пісочницях і заміна залежних сервісів на макети, які приймають запити по API та миттєво на них відповідають, що дозволяє окремо оцінювати продуктивність кожного мікросервісу. Для реалізації таких тестів було розроблено інструмент Terminus, який оцінює ємності мікросервісів на різних конфігураціях та використовує доречну регресійну модель до отриманих даних. Тестування на додатку, який складався з чотирьох мікросервісів, показало середню абсолютну відсоткову помилку менш 10%.

Стаття [45] описує Taiji – систему управління трафіком користувачів для широкомасштабних інтернет сервісів (Facebook). Головними її задачами є балансування навантаження на центри даних та мінімізація латентності запитів користувачів. Система контролює маршрутизацію між датацентрами та кордонними вузлами (edge nodes), які слугують у якості зворотних проксі та для кешування і розповсюдження статичного контенту. Taiji групує користувачів по зв’язкам, таким як слідування, дружба, тощо та перенаправляє трафік таких груп до одного центру даних. Система складається з двох комонентів: Runtime та Traffic Pipeline. Runtime визначає, яка частина трафіку

буде перенаправлена до доступних центрів даних для досягнення цілей рівню послуг, зазначених у заздалегіть визначеній політиці. Маршрутизація формулюється як задача про призначення, яка моделює обмеження та цілі оптимізації визначені сервісом. Результатом роботи компоненту є таблиця маршрутизації, що задовольняє політиці. Другий компонент – Traffic Pipeline – використовує цю таблицю та застосовує вищезазначене групування для генерації налаштувань маршрутизації для кожного кордонного балансиру навантажень. При цьому цей компонент не використовує цілі рівню послуг, а лише таблицю, згенеровану першим компонентом. Використання системи дозволило знизити навантаження на бекенд на 17% у порівнянні з Social Hash [46], що використовувалась раніше.

У [5] описується система автоматичного масштабування веб сервісів Facebook, яка основується на кількості роботи, яку потрібно виконати серверам у довільний момент часу. На відміну від утилізації, кількість роботи є абсолютною метрикою, яка не залежить від кількості серверів, що її виконують. Система використовує декілька параметрів для визначення необхідної кількості серверів. Спочатку історичні дані використовуються при тренуванні моделей машинного навчання для визначення закономірностей. При моделюванні ємності використовуються результати цих моделей. Наступним параметром є аварійний попит – тобто такий, який може виникнути у разі виходу з ладу найбільшого центру даних у регіоні. Окрім цього використовується поточний попит з деяким запасом, він порівнюється з завбаченим моделями попитом, та обирається найбільший. Щоб запобігти непередбачених ситуацій у випадку виходу зі строю системи, використовується додатковий сервіс Watchdog, який масштабує сервіси до безпечного рівня, зазвичай максимального за останні сім днів. Використання представленої системи дозволяє звільнювати ємність веб серверів для виконання іншої роботи під час обмеженого трафіку.

Авторами [47] описується Scryer – система прогностичного масштабування Netflix. Вона складається з чотирьох модулів: Data Collector, Predictor, Action Plan Generator та Scaler. Data Collector збирає дані з підключеного списку джерел, очищує та трансформує їх до наступного модулю. Predictor генерує прогнози у залежності від обраного алгоритму; представлено два алгоритми – на основі лінійної регресії та на основі швидкого перетворення Фур'є. Action Plan Generator використовує прогнози разом з іншими параметрами, такими як пропускна здатність серверів, для обчислення плану автоматичного масштабування. План оптимізується щодо мінімізації кількості подій масштабування при підтриманні оптимального розміру наступної партії. Scaler виконує згенерований план, включаючи складання розкладу масштабування, використовуючи AWS API. У якості метрики використовується кількість запитів користувачів за секунду.

Робота [48] презентує механізм кешування у контейнерній платформі Titus. Кеш було впроваджено на рівні API шлюзу (gateway). Система забезпечує узгодженість (consistency) за допомогою нового протоколу синхронізації, який використовує монотонні джерела часу та гарантує, що дані за запитом є на цей момент часу актуальними. Також описується ряд оптимізацій, таких як групування запитів перед синхронізацією та подача існуючих даних клієнтам, яких задовольняє кінцева узгодженість (eventual consistency).

У [49] представляється Kraken [50] – регістр Docker, що дозволяє розповсюджувати контейнери P2P. ПЗ підтримує різні системи сховища, такі як AWS S3 та HDFS. Заявлено підтримку більш 15000 хостів у кластері та теоретично безмежну масштабованість. Система розповсюдження будується на основі центрального компоненту – трекеру, який слідкує за наявністю контенту на вузлах та предоставляє списки вузлів для підключення, а передача даних організується самими вузлами. Підтримується аутентифікація та захист цілосності даних через TLS (Transport Layer Security).

1.2 Висновки за розділом

Масштабування додатків є актуальною та багатошаровою проблемою, що широко розглядається у сучасних джерелах.

Існуючі роботи можна умовно поділити на наступні категорії:

- Масштабування СУБД та БД
- Масштабування за допомогою мікросервісної архітектури
- Масштабування у хмарі
- Оптимізація балансирів навантажень
- Системи прогнозування навантаження
- Різноманітні комбінації вищенаведених

Споглядається тренд використання контейнерних систем, таких як Kubernetes, для полегшення процесу масштабування та розгортання додатків.

Багато компаній покладаються на готові хмарні рішення від AWS та Microsoft, а найкрупніші, такі як Meta (ex. Facebook), Netflix та Uber розроблюють свої системи у залежності від конкретних потреб. Деякі з цих систем були цілком опубліковані у вільному доступі, для інших було приведено принципи роботи, але ще більша кількість є пропрієтарними, та відомі лише розробникам.

У цілому проблема масштабування далека від вирішення та потребує подальшого досконального розглядання. Готові хмарні рішення являються особливо привабливими, т.я. вони дозволяють уникнути адміністрації ресурсів, тому задачею цієї роботи ставиться визначення придатності таких платформ для масштабування web додатків.

РОЗДІЛ 2. СЕРВІСИ БЕЗСЕРВЕРНИХ ОБЧИСЛЕНЬ ТА КЕРОВАНИХ БАЗ ДАНИХ

Хмарні сервіси суттєво змінили зберігання, управління та взаємодію з даними як для компаній, так і для приватних осіб. Локальні сервери та обладнання замінюються на глобально розподілені центри обробки даних, що надають обчислювальні ресурси на вимогу, що дозволяє масштабувати потужності, підвищувати надійність та знижувати витрати на обслуговування.

Amazon Web Services, разом з Microsoft Azure та Google Cloud є лідерами на ринку хмарної інфраструктури [51]. Усі три компанії збільшили темпи зростання порівняно з ситуацією на рік тому. Частки Amazon та Google на світовому ринку в третьому кварталі становили 31% і 13% відповідно, Microsoft Azure – склала 20%. Серед інших хмарних провайдерів найвищі темпи зростання мають Oracle, Huawei, Snowflake та Cloudflare. Основним фактором зросту ринку вважається генеративний ШІ (штучний інтелект).

Усі вищезазначені провайдери пропонують широкий вибір хмарних сервісів, багато з которих є прямими аналогами пропозицій конкурентів. Ці сервіси включають віртуальні машини, рішення для зберігання даних, бази даних, платформи для розробки додатків, аналітичні інструменти, фреймворки для машинного навчання,тощо.

Цінова політика, що дозволяє організаціям платити лише за використовувані ресурси, сприяє зменшенню початкових витрат та дозволяє уникати обслуговування інфраструктури, особливо при використанні так званих «безсерверних» платформ. Окрім економії коштів та масштабованості, хмара також сприяє швидкому розгортанню послуг та розширенню глобального охоплення.

Кожна з вищезазначених платформ має унікальні переваги, архітектурні принципи та спеціалізовані інструменти, розуміння яких є необхідним для прийняття обґрунтованих рішень у сучасному цифровому ландшафті.

2.1 Amazon Web Services

Платформа AWS була заснована компанією Amazon у 2006 році та до сих пір залишається найбільшою хмарною платформою.

На базовому рівні Amazon Elastic Compute Cloud (EC2) надає віртуалізовані сервери, які можна налаштовувати з різними процесорними архітектурами, обсягами пам'яті та рівнями мережевих навантажень. Вони можуть бути інтегровані з томами Amazon Elastic Block Store (EBS) та Amazon Elastic File System (EFS) зберігання даних. AWS підтримує контейнерні робочі навантаження через Amazon Elastic Container Service (ECS) або Amazon Elastic Kubernetes Service (EKS). Amazon Simple Storage Service (S3) пропонує об'єктне сховище з детальним контролем доступу, а Amazon Relational Database Service (RDS) та Amazon DynamoDB забезпечують керовані реляційні та NoSQL бази даних. Для великомасштабної аналітики даних пропонується Amazon Redshift у якості сховища петабайтів даних, а Amazon EMR інтегрується з Apache Hadoop і Spark для розподіленої обробки даних.

AWS управляє сотнями центрів обробки даних, розташованих у різних географічних районах, що допомагає клієнтам досягти низьку затримку, дотримуватися вимог до управління даними та забезпечувати безперервність бізнесу за допомогою стратегій резервування та відмовостійкості. Ці райони складаються з *регіонів* AWS і *зон доступності*.

Кожен регіон AWS – окрема географічна область, та є ізольованим від інших регіонів, що забезпечує максимальну відмовостійкість і стабільність. Автоматичної реплікації ресурсів між регіонами не проводиться. У таблиці 2.1 перелічено приклади доступних регіонів AWS. Регіони поділяються на кілька ізольованих зон, відомих як зони доступності. При розподіленні екземплярів між кількома зонами доступності, у випадку виходу з ладу одного з них підтримується обробка запитів екземпляром в іншій зоні доступності. Також підтримується використання еластичних IP-адресів, для маскування виходу з

ладу цього екземпляра шляхом швидкого перепризначення адреси на екземпляр в іншій зоні доступності.

Однією з основних переваг AWS є комплексна система безпеки, яка дозволяє використовувати цю платформу організаціям з високим рівнем державного контролю.

Будучи найбільш розповсюдженою платформою, AWS також пропонує обширну документацію, форуми користувачів, навчальні програми та сертифікації, та позиціонує себе як провайдера, що може забезпечити будь які хмарні вимоги.

Таблиця 2.1 –Регіони AWS

Ідентифікатор	Ім’я
us-east-1	US East (N. Virginia)
us-west-1	US West (N. California)
af-south-1	Africa (Cape Town)
ap-east-1	Asia Pacific (Hong Kong)
ap-south-1	Asia Pacific (Mumbai)
ca-central-1	Canada (Central)
ca-west-1	Canada West (Calgary)
cn-north-1	China (Beijing)
cn-northwest-1	China (Ningxia)
eu-central-1	Europe (Frankfurt)
eu-south-1	Europe (Milan)
il-central-1	Israel (Tel Aviv)
me-south-1	Middle East (Bahrain)
me-central-1	Middle East (UAE)
sa-east-1	South America (São Paulo)

AWS орієнтована на організації, яким потрібні масштабовані, настроювані рішення і широкий спектр хмарних сервісів. Сюди входять компанії, що займаються електронною комерцією, такі як Shopify, які використовують надійний хостинг і гнучку масштабованість AWS; платформи потокового мультимедіа, як Netflix, що користуються перевагами мережі доставки контенту (CDN) і глобальних центрів обробки даних AWS; а також компанії, що обробляють великі обсяги даних, що використовують інструменти аналітики та озера даних AWS, такі як Redshift і Athena.

2.1.1 AWS Lambda

AWS Lambda – це керований подіями, безсерверний сервіс / функція як сервіс (FaaS – Function as a Service), що надається Amazon Web Services. Він виконує код у відповідь на події та автоматично керує обчислювальними ресурсами, необхідними для цього коду, включаючи обслуговування сервера і операційної системи, виділення потужностей і автоматичне масштабування, а також ведення журналів. Lambda запускає функції за необхідністю і масштабує їх автоматично [52].

Lambda пропонується як сервіс для додатків, які потребують швидкого масштабування у залежності від попиту та може використовуватися для:

- обробки файлів: разом з Amazon Simple Storage Service (Amazon S3), Lambda може запустити обробку даних в режимі реального часу після завантаження;
- обробки потоків: разом з Amazon Kinesis потокові дані можуть оброблюватись у реальному часі для відстеження активності додатків, обробки транзакційних замовлень, аналізу потоку кліків, очищення даних, фільтрації журналів, індексації, аналізу соціальних мереж, телеметрії даних пристроїв Інтернету речей (IoT), тощо;
- веб-додатків: у поєднанні з іншими сервісами AWS, Lambda може використовуватися для створення потужних веб-додатків, які автоматично масштабуються і працюють у високодоступній конфігурації у декількох центрах обробки даних;
- бекендів Інтернету речей: безсерверні бекенди Lambda можуть використоватися для обробки веб, мобільних, IoT та сторонніх API-запитів;
- мобільних бекендів: з використанням Amazon API Gateway Lambda може використовуватися для автентифікації та обробки API-запитів.

Оскільки Lambda керує цими ресурсами, можливість налаштування операційної системи на наданих середовищах виконання відсутня. Lambda

виконує операційні та адміністративні дії від імені розробника, включаючи управління ресурсами, моніторинг та реєстрацію інших функцій Lambda.

Lambda розподіляє потужність процесора пропорційно до обраного обсягу пам'яті, доступного окремій Lambda функції під час виконання. Пам'ять налаштовується у діапазоні від 128 МіБ до 10 240 МіБ з кроком у 1 МіБ. При 1,769 МіБ функція має еквівалент одного vCPU (virtual CPU – віртуальний центральний процесор).

Основні можливості Lambda:

- змінні середовища, що можуть використовуватися для налаштування поведінки функції без оновлення коду;
- версії – надається підтримка версій для тестування без впливу на стабільну виробничу версію;
- образи контейнерів – контейнери для функції Lambda, можна створювати використовуючи як базовий образ, наданий AWS, так і альтернативн;
- шари, які дозволяють пакування бібліотек та інших залежностей, для зменшення розміру архівів розгортання та прискорення розгортання;
- розширення Lambda – предоставляються інструменти для моніторингу, спостереження, безпеки та управління;
- URL функцій – підтримуються виділені HTTP(S) ендпойнти до функцій Lambda;
- стрімінг відповіді – підтримується стрімінг відповідей клієнтам з функцій Node.js, для покращення часу до першого байту (TTFB) або повернення більших обсягів даних;
- контроль паралельності та масштабування – предоставляється детальний контроль над масштабуванням та швидкістю відклику;
- підпис коду – дозволяє перевіряти кожне розгортання коду та підтверджувати, що пакети коду підписані надійними джерелами;

- приватні мережі – надається підтримка приватних мереж для ресурсів, таких як бази даних, внутрішні сервіси, тощо;
- доступ до файлової системи – дозволяє приєднання Amazon Elastic File System (Amazon EFS) до локального каталогу для доступу до спільних ресурсів;
- Lambda SnapStart для Java – покращує швидкість запуску середовищ Java до 10 разів без додаткових витрат, зазвичай без змін у коді функції.

AWS Lambda підтримує декілька мов завдяки використанню середовищ виконання (runtime). Середовище виконання забезпечує специфічне для мови середовище, яке передає події виклику, контекстну інформацію та відповіді між Lambda та функцією. Надається можливість використовувати середовища виконання, які надаються Lambda, або створити власні.

AWS Lambda підтримує певний набір мов (Node.js, Python, Java, Go, Ruby, .NET), а також пропонує користувацькі середовища виконання (табл. 2.3), проте такі середовища мають базуватися на Amazon Linux без можливості виконання функцій на базі Windows. Lambda вимагає певний формат пакування (ZIP-архів або образ контейнера) і не дозволяє створювати повністю довільне середовище операційної системи.

У Таблиці 2.2 перелічено підтримувані середовища виконання Lambda та прогнозовані дати застарівання (deprecation). Після застарівання середовища виконання створення та оновлення функцій підтримується лише протягом обмеженого періоду часу.

AWS Lambda стягує плату на основі кількості запитів і тривалості виконання, а також об’єму обраної виділеної пам'яті. У залежності від обраного регіону та архітектури процесору (arm або x86), вартісь 1000000 запитів може становити від 0,18 до 0.28 USD, а вартість одного GiB/c – від 0,0000096 до 0,00002292 USD.

Початковий запуск функції може займати велику кількість часу на ініціалізацію, тому Lambda пропонує можливість підтримки завжди «теплих» екземплярів функції (provisioned concurrency). У цьому випадку, окрім плати за запити та тривалість виконання, знімається додаткова плата за час підтримки таких теплих екземплярів.

Таблиця 2.2 – Доступні середовища виконання Lambda

Ім’я	Ідентифікатор	Операційна система	Дата застарівання
Node.js 20	nodejs20.x	Amazon Linux 2023	Не заплановано
Node.js 18	nodejs18.x	Amazon Linux 2	31-07-2025
Python 3.13	python3.13	Amazon Linux 2023	Не заплановано
Python 3.12	python3.12	Amazon Linux 2023	Не заплановано
Python 3.11	python3.11	Amazon Linux 2	Не заплановано
Python 3.10	python3.10	Amazon Linux 2	Не заплановано
Python 3.9	python3.9	Amazon Linux 2	Не заплановано
Java 21	java21	Amazon Linux 2023	Не заплановано
Java 17	java17	Amazon Linux 2	Не заплановано
Java 11	java11	Amazon Linux 2	Не заплановано
Java 8	java8.al2	Amazon Linux 2	Не заплановано
.NET 8	dotnet8	Amazon Linux 2023	Не заплановано
.NET 6	dotnet6	Amazon Linux 2	20-12-2024
Ruby 3.3	ruby3.3	Amazon Linux 2023	Не заплановано
Ruby 3.2	ruby3.2	Amazon Linux 2	Не заплановано
OS-only Runtime	provided.al2023	Amazon Linux 2023	Не заплановано
OS-only Runtime	provided.al2	Amazon Linux 2	Не заплановано

Lambda встановлює ліміти на обсяг обчислювальних ресурсів і ресурсів зберігання, які можна використовувати для запуску і зберігання функцій. Ліміти на одночасне виконання та зберігання застосовуються для кожного регіону AWS (табл. 2.3). Багато з цих лімітів можна збільшити через окремий запит у службу підтримки AWS, проте цей процес не відрізняється прозорістю та займає певний час, що може викликати неочікувані затримки. Також, ліміти для нових облікових записів AWS є меншими від заявлених у офіційній документації.

Таблиця 2.3 – Ліміти ресурсів AWS Lambda

Ресурс	Ліміт
Паралельні виконання	1000, можна збільшити до десятків тисяч
Сховище для завантажених функцій (архіви .zip) і шарів.	75 ГБ, можна збільшити до терабайт
Еластичні мережеві інтерфейси VPC	500, можна збільшити до тисяч
Пам'ять функції	128 МіБ до 10 240 МіБ
Таймаут функції	900 секунд
Змінні середовища функції	4 КБ
Шари функції	5 шарів
Ліміт масштабування паралелізму функцій	1,000 середовищ виконання кожні 10 секунд
Навантаження виклику (запит та відповідь)	6 МіБ на запит і відповідь (синхронно) 20 МіБ для кожної відповіді (синхронно) 256 КБ (асинхронно)
Пропускна здатність для потокових відповідей	Необмежений для перших 6 МіБ відповіді 2 МіБ/с для решти відповіді
Розмір пакета розгортання (архіву .zip)	50 МіБ (заархівовані, при завантаженні через Lambda API або SDK). 50 МіБ (при завантаженні через консоль)
Розмір налаштувань образу контейнера	16 КБ
Розмір пакету коду образу контейнера	10 ГБ (максимальний розмір зображення без стиснення, включно з усіма шарами)
Тестові події (редактор консолі)	10
Каталог /tmp	Від 512 МіБ до 10 240 МіБ
Дескриптори файлів	1,024
Виконавчі процеси/потоки	1,024
Запити на виклик для кожної функції на регіон (синхронні)	Кожен екземпляр середовища виконання може обслуговувати до 10 запитів на секунду.
Запити на виклик для кожної функції на регіон (асинхронні)	Кожен екземпляр середовища виконання може обслуговувати необмежену кількість запитів.

Таким чином AWS Lambda пропонує масштабований, керований подіями обчислювальний сервіс, який дозволяє розробникам виконувати код без виділення ресурсів або керування серверами.

2.1.2 Amazon Aurora

Amazon Aurora (Aurora) — це повністю керована реляційна база даних, яка сумісна з MySQL і PostgreSQL [53]. Aurora є частиною сервісу керованих баз даних Amazon Relational Database Service (Amazon RDS). Amazon RDS спрощує налаштування, експлуатацію та масштабування реляційних баз даних у хмарі.

З деякими робочими навантаженнями Aurora може забезпечити до п'яти разів більшу пропускну здатність ніж MySQL, і до трьох разів більшу пропускну здатність ніж PostgreSQL не вимагаючи змін у більшості існуючих додатків.

Aurora включає в себе високопродуктивну підсистему зберігання даних. Базове сховище автоматично збільшується в міру необхідності. Обсяг кластера Aurora може зростати до максимального розміру 128 тебібайт (TiB). Aurora також автоматизує та стандартизує кластеризацію та реплікацію баз даних, які зазвичай є одними з найскладніших аспектів конфігурації та адміністрування баз даних.

При створенні нових серверів баз даних через Amazon RDS надається вибір між Aurora MySQL або Aurora PostgreSQL. Aurora використовує функції Amazon Relational Database Service (Amazon RDS) для керування та адміністрування. Aurora використовує інтерфейс Amazon RDS AWS Management Console, команди AWS CLI (Command Line Interface – інтерфейс командної строки) та операції API для виконання рутинних завдань з базами даних, таких як резервне копіювання, відновлення, виявлення збоїв, тощо. Операції управління Aurora зазвичай включають цілі кластери серверів баз даних, які синхронізуються за допомогою реплікації, а не окремі екземпляри баз даних. Автоматична кластеризація, реплікація та розподіл сховища спрощують налаштування, експлуатацію та масштабування найбільших розгортань MySQL та PostgreSQL.

Перенос даних з Amazon RDS для MySQL і Amazon RDS для PostgreSQL в Aurora виконується за допомогою створення і відновлення знімків або налаштування односторонньої реплікації.

Кластер бази даних Amazon Aurora складається з одного або декількох екземплярів баз даних і тома кластера, який керує даними для цих екземплярів баз даних. Том кластера Aurora – це віртуальний том сховища бази даних, який охоплює кілька зон доступності, причому кожна зона доступності має копію даних кластера DB. Кластер Aurora складається з двох типів екземплярів БД:

- первинний (записуючий) екземпляр БД – підтримує операції читання і запису, а також виконує всі модифікації даних в томі кластера. Кожен кластер Aurora DB має один первинний екземпляр БД;
- репліка Aurora (екземпляр БД для читання) – підключається до того самого тома сховища, що й основний екземпляр БД, але підтримує лише операції читання. Кожен кластер Aurora DB може мати до 15 реплік Aurora Replicas додатково до основного екземпляра бази даних. Aurora автоматично перемикається на репліку Aurora, якщо основний екземпляр бази даних стає недоступним. Репліки Aurora також можуть розвантажувати робочі навантаження на читання з основного екземпляра БД.

Наступна діаграма ілюструє взаємозв'язок між сховищем даних, записуючим екземпляром БД, та екземплярами БД для читання у кластері Aurora DB (Рис 2.1).

Кластер Aurora DB ілюструє розділення обчислювальних потужностей і сховища. Наприклад, конфігурація Aurora з одним екземпляром БД всеодно є кластером, оскільки базовий обсяг сховища включає декілька вузлів зберігання, розподілених по декількох зонах доступності (AZ – Availability Zones).

Операції вводу/виводу (I/O) у кластерах Aurora DB рахуються незалежно від того, на якому екземплярі БД вони виконуються – на записуючому або екземплярі для читання.

Кожна версія Amazon Aurora сумісна з певною версією бази даних MySQL або PostgreSQL.

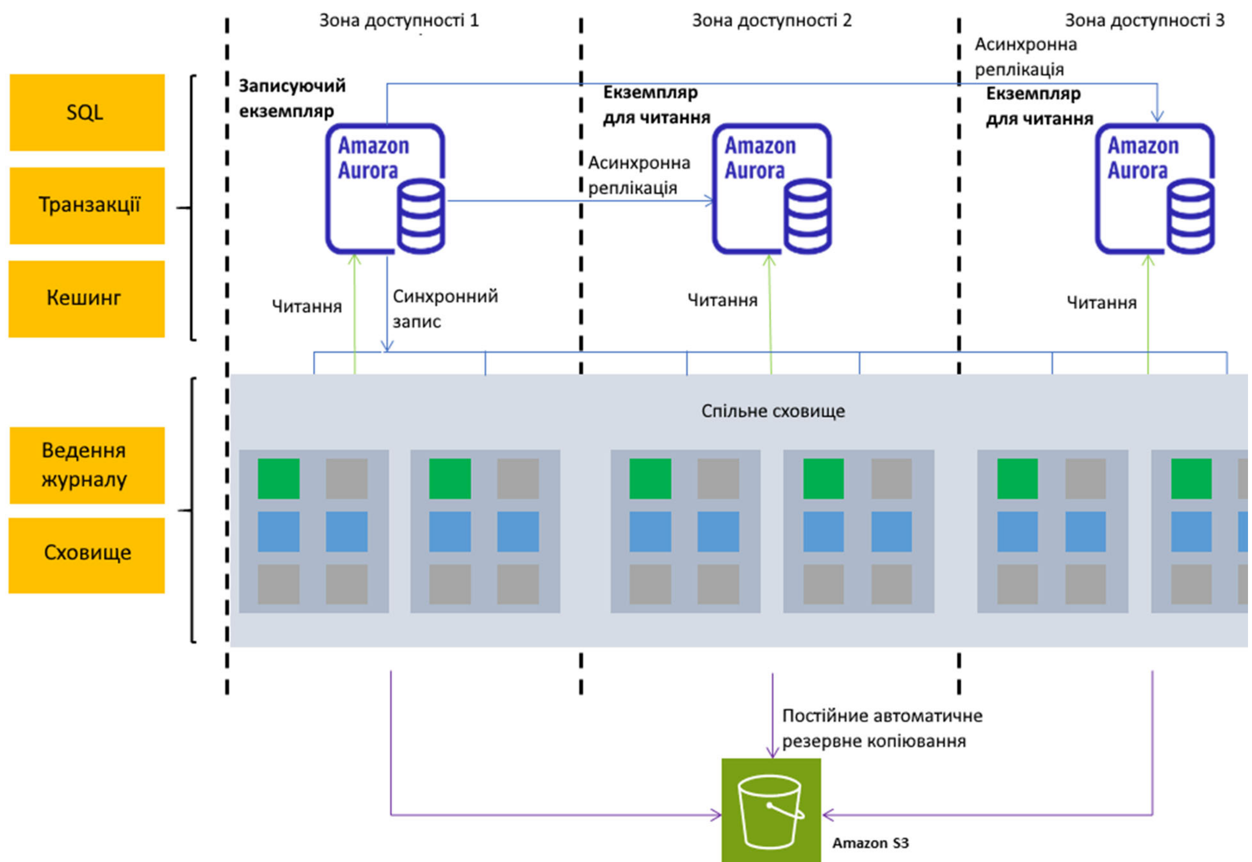

Рис. 2.1 – Кластер Aurora

Зона доступності – це ізольована зона в певному регіоні AWS. У випадку створення VPC (Virtual Private Cloud / приватна віртуальна хмара) і підмереж замість використання VPC за замовчуванням, AZ вказується для кожної підмережею. При створенні кластеру Aurora DB, Aurora створює первинний екземпляр в одній з підмереж у групі підмереж VPC. Таким чином, цей екземпляр пов'язується з конкретною AZ, обраною Aurora.

Кожен кластер баз даних Aurora розміщує копії свого сховища у трьох окремих AZ, автоматично вибраних Aurora з числа AZ у групі підмереж бази даних. Кожен екземпляр бази даних у кластері повинен знаходитися в одній з цих трьох AZ. При створенні екземпляру бази даних у кластері, Aurora

автоматично обирає відповідну AZ для цього екземпляра, якщо вона не вказана окремо.

Клас екземпляра БД визначає обчислювальну потужність і обсяг пам'яті екземпляра БД Amazon Aurora. Вибір класу екземпляру бази даних залежить від потрібної обчислювальної потужності та вимог до пам'яті. Клас екземпляра БД складається з типу та розміру. Amazon Aurora підтримує класи екземплярів БД для наступних випадків використання:

- Aurora Serverless v2 (безсерверний);
- Memory-optimized (опитмізований по пам’яті);
- Burstable-performance (зі змінною продуктивністю);
- Optimized Reads (оптимізований для читання).

Aurora зберігає дані у кластерному томі, котрий є єдиним віртуальним томом, що використовує твердотільні накопичувачі (SSD). Кластерний том складається з копій даних у трьох зонах доступності в одному регіоні AWS. Том кластера Aurora містить усі користувацькі дані, об'єкти схеми та внутрішні метадані, такі як системні таблиці та журнал. Архітектура Aurora зі спільним сховищем робить дані незалежними від екземплярів БД у кластері.

Amazon Aurora підтримує дві конфігурації кластерного сховища баз даних:

- Aurora I/O-Optimized – покращена продуктивність для додатків з інтенсивним записом та читанням. Плата знимається лише за використання та зберігання кластерів баз даних, без додаткових витрат на операції читання та запису.Aurora I/O-Optimized рекомендується тоді, коли витрати на операції вводу-виводу становлять 25% або більше від загальних витрат на базу даних Aurora;
- Aurora Standard – передзначена для додатків з обмеженим використанням операцій запису та читання. На додаток до використання та зберігання кластерів баз даних, плата знімається за 1 мільйон запитів на операції вводу-виводу.

Томи кластерів Aurora автоматично зростають зі збільшенням обсягу даних у вашій базі даних. Максимальний розмір тома кластера Aurora становить 128 тебібайт (TiB) або 64 TiB, залежно від версії БД.

Aurora підтримує декілька варіантів реплікації. Кожен кластер БД Aurora має вбудовану реплікацію між декількома екземплярами БД в одному кластері. При створенні другого, третього і т.д. екземпляру бази даних у кластері Aurora, Aurora автоматично налаштовує реплікацію з записуючого екземпляра до всіх інших екземплярів бази даних. Ці інші екземпляри БД доступні лише для читання і називаються репліками Aurora.

Репліки Aurora мають два основних призначення: розподілення навантаження для читання та перехоплення ролі записуючого екземпляру в кластері у випадку його недоступності. Кластер Aurora DB може містити до 15 реплік Aurora. Репліки Aurora можуть бути розподілені між зонами доступності кластеру БД. Основний екземпляр і репліки Aurora в кластері БД бачать дані в томі кластера як єдиний логічний том. В результаті всі репліки Aurora повертають однакові дані з мінімальною затримкою між репліками.

Плата за екземпляри Amazon RDS у кластері Amazon Aurora знімається на основі наступних компонентів:

- години роботи екземпляра БД;
- обсяг сховища, наданого екземпляру БД;
- кількість запитів на ввід/вивід;
- резервне сховище – для автоматизованих резервних копій баз даних і будь-яких створених активних знімків баз даних;
- передача даних до та з екземпляра БД з інтернету та інших регіонів AWS.

Amazon Aurora підтримує конфігурацію з автоматичним масштабуванням на вимогу, яка має назву Aurora Serverless v2, яка допомагає автоматизувати процеси моніторингу робочого навантаження та налаштування ємності баз даних. Обсяг пам'яті налаштовується автоматично

на основі попиту додатків. Плата знімається лише за ресурси, які споживають кластери баз даних. Використовуючи Aurora Serverless v2, можна створити кластер Aurora DB та вказати мінімальний і максимальний діапазон продуктивності. Aurora масштабує кожен записник або зчитувач Aurora Serverless v2 в кластері в межах цього діапазону. Aurora Serverless v2 автоматично масштабує ресурси бази даних на основі мінімальних і максимальних специфікацій. У Aurora Serverless v2, як і в кластерах з виділеними ресурсами, ємність сховища та обчислювальна потужність розділені. При описі масштабування Aurora Serverless v2 вказується обчислювальна потужність, яка збільшується або зменшується. Таким чином, кластер може містити великий обсяг даних, навіть коли процесор і обсяг пам'яті зменшуються до низьких рівнів

Одиницею виміру для Aurora Serverless v2 є Aurora capacity unit (ACU), які можна налаштовувати у межах 0.5 – 256, у залежності від версії БД. Кожна ACU – це комбінація приблизно 2 гігабайт пам'яті, відповідного процесора та пропускної можливості мережі. Для кожного записувача або зчитувача Aurora Serverless v2 Aurora безперервно відстежує використання ресурсів, таких як процесор, пам'ять і мережа. Ці параметри в сукупності називаються навантаженням. Навантаження включає в себе операції з базою даних, що виконуються додатком. Воно також включає фонові операції сервера бази даних і адміністративні завдання Aurora. Коли пропускна здатність обмежена будь-яким з цих факторів, Aurora Serverless v2 автоматично масштабується.

2.2 Microsoft Azure

Azure – платформа хмарних обчислень від Microsoft, що пропонує широкий спектр послуг для розробки додатків, управління даними, аналітики та машинного навчання. Її базові інфраструктурні сервіси базуються на віртуальних машинах Azure та сервісі Azure Kubernetes, які дозволяють створювати та організовувати віртуалізовані обчислювальні екземпляри, що працюють під управлінням різних операційних систем та контейнерних робочих навантажень. Додатково, пропонуються спеціалізовані служби даних, такі як Azure SQL Database та Azure Cosmos DB, які надають керовані реляційні та NoSQL бази даних, оптимізовані для глобального розповсюдження та горизонтального масштабування, а також Azure Synapse Analytics, яка може оброблювати великі об’єми даних та має вбудовану підтримку SQL-запитів і моделей машинного навчання. Azure Machine Learning пропонує платформу для навчання, розгортання та управління великомасштабними моделями машинного навчання, включаючи автоматизоване налаштування гіперпараметрів та інтеграцію з обчислювальними кластерами. Наукова актуальність платформи обумовлена підтримкою робочих навантажень для високопродуктивних обчислень за допомогою кластерів на базі InfiniBand і графічних процесорів, які полегшують симуляції в гідродинаміці, молекулярному моделюванні, тошо. Аналогічно, служби Інтернету речей Azure, такі як Azure IoT Hub і Azure Digital Twins, надають засоби для збору, моделювання та обробки даних з датчиків в режимі реального часу.

Глобальна інфраструктура Azure розподілена по багатьох географічних регіонах, що дозволяє зменшувати затримку для кінцевих користувачів та задовольняти вимогам щодо розміщення даних. *Регіон* Azure – набір центрів обробки даних, розгорнутих у межах периметра з визначеною затримкою і

з'єднаних через виділену регіональну мережу. Приклади регіонів можна побачити у таблиці 2.4.

Таблиця 2.4 –Регіони Microsoft Azure

Ідентифікатор	Ім’я
Australia Central	Канберра
Brazil South	Сан-Паулу
Central India	Пуна
Central US	Айова
Chile Central	Сантьяго
China East	Шанхай
East Asia	Гонконг
East US	Вірджинія
Indonesia Central	Джакарта
Israel Central	Ізраїль
Japan East	Токіо, Сайтама
Korea Central	Сеул
North Europe	Ірландія
Southeast Asia	Сінгапур
Taiwan North	Тайбей
UAE North	Дубай
UK South	Лондон
West Europe	Нідерланди
West US	Каліфорнія

У багатьох регіонах Azure передбачені зони доступності, які є окремими групами центрів обробки даних у регіоні та мають незалежне живлення, охолодження та мережеву інфраструктуру.

Екосистема платформи включає різноманітні інструменти для розробки та DevOps, такі як Visual Studio, GitHub Actions та Azure DevOps, що спрощують цикл розробки програмного забезпечення.

Azure має тісну інтеграцію з екосистемою Microsoft, що облегшує її використання у організацях, які використовують Windows Server, Active Directory та Microsoft Office та спрощує впровадження гібридних хмарних рішень або перехід з локальної системи в хмару. Також платформа відповідає галузевим нормам і стандартам (наприклад, HIPAA, GDPR), що є критичним фактором для організацій, які працюють з конфіденційними даними.

2.2.1 Azure Functions

Azure Functions – це хмарний сервіс, доступний за запитом, який предоставляє інфраструктуру та ресурси, необхідні для запуску програм [54]. Azure Functions – це служба безсерверних обчислень / FaaS, що надається Microsoft Azure, яка дає змогу розробникам виконувати код у відповідь на події, без необхідності керування інфраструктурою. Вона дозволяє виконувати невеликі фрагменти коду, які називаються «функціями», у відповідь на різні тригери, такі як HTTP-запити, таймери або повідомлення від інших служб Azure, таких як Azure Queue Storage або Azure Event Hubs. Azure Functions автоматично масштабується відповідно до попиту, стягуючи плату лише за витрачений обчислювальний час.

Azure Functions надають набір тригерів і прив'язок на основі подій, які пов'язують функції з іншими сервісами без необхідності написання додаткового коду, та можуть використовуватися для, поміж інших, наступних сценаріїв:

- обробки завантажених файлів;
- обробки даних в режимі реального часу;
- обробки моделей даних;
- запуску завданнь за розкладом;
- побудови масштабованиих веб api;
- побудови безсерверних робочих процесів;
- відповідей на зміни в базах даних;
- створення систем повідомлень.

Azure Functions надає підтримку для розробки на C#, Java, JavaScript, PowerShell, Python, а також можливість використовувати інші мови, такі як Rust і Go. Сервіс інтегрується з Visual Studio, Visual Studio Code, Maven та іншими популярними інструментами розробки.

Azure Functions також інтегрується з Azure Monitor і Azure Application Insights, для забезпечення телеметрії під час виконання та аналізу функцій у хмарі.

При створенні функції в Azure, підтримується декілька варіантів хостингу (таблиця 2.5), від яких залежить наступна поведінка:

- яким чином функція масштабується;
- ресурси, що доступні для кожного екземпляра функції;
- підтримка розширених функціональних можливостей, таких як підключення до віртуальної мережі Azure;
- підтримка контейнерів Linux.

Таблиця 2.5 – Варіанти хостингу Azure Functions

Варіант	Сервіс	Опис
Flex Consumption plan	Azure Functions	Функції динамічно додаються та видаляються у залежності від налаштувань та кількості подій. Підтримуються заздалегідь готові (теплі), завжди активні функції. Підтримується віртуальна мережа. Масштабується автоматично. Оплата лише за активні функції.
Premium plan	Azure Functions	Автоматично масштабується на основі попиту за допомогою попередньо розігрітих (теплих) робочих станцій, які запускають програми без затримок, працюють на потужніших машинах і підключаються до віртуальних мереж.
Dedicated plan	Azure Functions	Функції працюють у рамках плану App Service за звичайними тарифами плану App Service. Найкраще підходить для довготривалих сценаріїв.
Container Apps	Azure Container Apps	Функції виконуються у повністю керованому середовищі, розміщеному в Azure Container Apps.
Consumption plan	Azure Functions	Автоматичне масштабування з оплатою лише активних функцій, але без додаткових сервісів Flex Consumption plan.

У залежності від обраного плану, Azure Functions мають наступні обмеження (Таблиця 2.6)

Таблиця 2.6 – Обмеження Azure Functions у залежності від обраного плану

Ресурс / План	Flex Consumption plan	Premium plan	Dedicated plan/ASE	Container Apps	Consumption plan
Таймаут за замовчуванням	30 хв	30 хв	30 хв	30 хв	5 хв
Максимальний таймаут (хв)	-	-	-	-	10
Максимум вихідних з'єднань	-	-	-	-	600 активних (1200 разом)
Максимальний запит (МіБ)	210	210	210	210	210
Максимальна довжина рядка запиту	4096	4096	4096	4096	4096
Максимальна довжина URL запиту	8192	8192	8192	8192	8192
ACU на екземпляр	210-840	100-840/ 210-250	змінний	100	змінний
Максимальна пам'ять (ГБ)	4	3.5-14	1.75-256/ 8-256	змінний	1.5
Максимум екземплярів (Windows/Linux)	100/20	визнач. SKU/100	10-300	200/100	1000
Функціональні додатки	100	100	-	-	100
Плани App Service	n/a	100 на групу	100 на групу	n/a	100 на регіон
Слоти розгортання	n/a	3	1-20	n/a	2
Сховище (тимчасове)	0.8 GB	21-140 GB	11-140 GB	n/a	0.5 GB
Сховище (постійне)	0 GB7	250 GB	10-1000 GB11	n/a	1 GB
Користувацькі домени	500	500	500	n/a	500

Azure Functions потребують обліковий запис сховища, пов'язаний з кожною функцією. Він використовується хостом функцій для таких операцій, як керування тригерами та ведення журналу виконання функцій, а також для динамічного масштабування функціональних програм.

Azure Functions підтримує широкий спектр середовищ програмування (табл. 2.7) і може працювати як на хостах Linux, так і Windows, залежно від обраного тарифного плану.

Таблиця 2.7 – Підтримка мов програмування Azure Functions

Мова	Версія	Рівень підтримки	Дата застарівання
C#	.NET 9	Попередній	12-05-2026
C#	.NET 8	Загальний	10-11-2026
C#	.NET 6	Загальний	12-11-2024
C#	.NET Framework 4.8.1	Загальний	-
Java	Java 21 (Linux)	Попередній	09-2028
Java	Java 17	Загальний	09-2027
Java	Java 11	Загальний	09-2027
Java	Java 8	Загальний	30-11-2026
JavaScript	Node.js 22	Попередній	30-04-2027
JavaScript	Node.js 20	Загальний	30-04-2026
JavaScript	Node.js 18	Загальний	30-04-2025
PowerShell	PowerShell 7.4	Загальний	10-11-2026
PowerShell	PowerShell 7.2	Загальний	08-11-2024
Python	Python 3.11	Загальний	10-2027
Python	Python 3.10	Загальний	10-2026
Python	Python 3.9	Загальний	10-2025
Python	Python 3.8	Загальний	10-2024
TypeScript	Node.js 22	Попередній	30-04-2027
TypeScript	Node.js 20	Загальний	30-04-2026
TypeScript	Node.js 18	Загальний	30-04-2025

У цілому Azure Functions пропонує безсерверну обчислювальну платформу для виконання керованого подіями коду, що забезпечує автоматизоване масштабування та інтеграцію з хмарною екосистемою Microsoft.

2.2.2 Azure SQL Database

Azure SQL – це сімейство керованих продуктів, які використовують механізм баз даних SQL Server у хмарі Azure [55].

Azure SQL охоплює три продукти :

- Azure SQL Database: Підтримує сучасні хмарні додатки на основі інтелектуальної керованої служби баз даних, що включає безсерверні обчислення;
- Azure SQL Managed Instance: Масштабована служба хмарних баз даних, яка завжди працює на базі останньої стабільної версії механізму баз даних Microsoft SQL Server і операційної системи з 99,99% вбудованою доступністю та забезпечує майже 100% сумісність з SQL Server;
- SQL Server у віртуальних машинах Azure: Дозволяє використовувати повні версії SQL Server у хмарі без необхідності керувати локальним обладнанням.

Хоча платформа Azure підтримує повністю керовані сервіси для MySQL та PostgreSQL, вони не підтримують автоматичного масштабування, тому у даній роботі розглядатися не будуть.

Azure SQL Database – це повністю керована платформа як послуга (PaaS), яка виконує більшість функцій керування базами даних, таких як оновлення, виправлення, резервне копіювання та моніторинг без участі користувача. Вона дозволяє легко визначати і масштабувати продуктивність в рамках двох різних моделей: на основі vCore (virtual core – віртуальний ЦП) і на основі DTU (Database transaction units – одиниці транзакції БД). Модель придбання на основі vCore дає змогу вибрати кількість ядер vCore, обсяг пам'яті, а також обсяг і швидкість зберігання даних. Модель придбання на основі DTU пропонує поєднання обчислювальних ресурсів, пам'яті та ресурсів вводу/виводу на трьох рівнях обслуговування для підтримки різних робочих навантажень баз даних. Обсяги обчислень на кожному рівні забезпечують

різний набір цих ресурсів, до яких можна додати додаткові ресурси зберігання даних.

Модель придбання на основі vCore передбачає два різні обчислювальні рівні бази даних Azure SQL – рівень виділених обчислень і рівень безсерверних обчислень. Модель придбання на основі DTU передбачає лише рівень виділених обчислень.

Рівень виділених обчислювальних ресурсів надає певний обсяг обчислювальних ресурсів, який постійно надається незалежно від активності робочого навантаження, і виставляє рахунки за обсяг наданих обчислювальних ресурсів за фіксованою ціною за годину.

Рівень безсерверних обчислень автоматично масштабує обчислювальні ресурси залежно від активності робочого навантаження та виставляє рахунки за обсяг використаних обчислень за секунду.

Кожна окрема база даних ізольована від інших та має власний гарантований обсяг обчислювальних ресурсів, пам'яті та сховища. Обсяг ресурсів призначено для цієї бази даних і не групується спільно з іншими базами даних в Azure. Ресурси однієї бази даних можна динамічно масштабувати вгору та вниз. Наприклад, можна отримати від 1 до 128 vCore або від 32 ГБ до 4 ТБ. Рівень послуг Hyperscale дозволяє масштабувати до 128 ТБ з можливістю швидкого резервного копіювання та відновлення.

За допомогою еластичних пулів можна призначати ресурси, які спільно використовуються всіма базами даних у пулі. Надається можливість створити нову базу даних або перемістити існуючі окремі бази даних в пул ресурсів, щоб максимально ефективно використовувати ресурси. Ця опція також надає можливість динамічно масштабувати ресурси еластичного пулу вгору і вниз.

Динамічне масштабування у цьому контексті відрізняється від автомасштабування. Автомасштабування – служба масштабується автоматично на основі критеріїв, тоді як динамічне масштабування дозволяє масштабувати вручну без простоїв. Окрема база даних підтримує динамічне

масштабування, але не автомасштабування. Для більш автоматичного масштабування пропонуються наступні альтернативи:

- використання безсерверного рівня, який забезпечує автоматичне масштабування;
- використання скриптів для планування або автоматизації масштабування для окремої бази даних;
- використання еластичних пулів, які дозволяють базам даних спільно використовувати ресурси в пулі відповідно до індивідуальних потреб кожної бази даних. Еластичні пули також можна масштабувати за допомогою спеціальних сценаріїв, що дозволяє планувати або автоматизувати масштабування.

Модель придбання на основі vCore пропонує три рівні обслуговування:

- загальний рівень послуг, призначений для типових робочих навантажень, який пропонує бюджетні збалансовані варіанти обчислень і зберігання даних;
- Business Critical, що призначений для додатків з високою швидкістю транзакцій та строгими вимогами до затримок вводу/виводу. Він пропонує найвищу стійкість до збоїв завдяки використанню декількох ізольованих реплік;
- гіпермасштабний рівень обслуговування призначений для більшості бізнес-навантажень. Він забезпечує велику гнучкість і високу продуктивність завдяки незалежно масштабованим обчислювальним ресурсам і ресурсам зберігання та пропонує вищу стійкість до збоїв, дозволяючи налаштовувати більш однієї репліки бази даних.

Модель придбання на основі DTU пропонує два рівні обслуговування:

- стандартний рівень обслуговування призначений для звичайних робочих навантажень. Він пропонує бюджетні збалансовані варіанти обчислень та зберігання даних;

- преміум-рівень послуг призначений для OLTP-додатків з високою швидкістю транзакцій і низькими вимогами до затримок вводу/виводу. Він пропонує найвищу стійкість до збоїв завдяки використанню декількох ізольованих реплік.

У Azure SQL пропонується три архітектурні моделі доступності:

- модель віддаленого зберігання, яка базується на розділенні обчислень і зберігання даних. Вона покладається на доступність і надійність рівня віддаленого зберігання. Ця архітектура орієнтована на бюджетні бізнес-додатки, які можуть терпіти деяке зниження продуктивності під час технічного обслуговування.

- локальна модель зберігання, яка базується на кластері процесів механізму бази даних. Ця архітектура орієнтована на критично важливі додатки з високою продуктивністю вводу-виводу та високою швидкістю транзакцій.

- гіпермасштабна (Hyperscale) модель (Рис. 2.2), яка використовує розподілену систему високодоступних компонентів, таких як обчислювальні вузли, сервери сторінок, служба журналів та постійне сховище. Кожен компонент, що підтримує таку базу даних, забезпечує власну стійкість до збоїв. Обчислювальні вузли, сервери сторінок і служба журналів працюють на Azure Service Fabric, яка контролює працездатність кожного компонента і виконує перемикання на доступні справні вузли в разі потреби. Для постійного сховища використовується Azure Storage.

У кожній з трьох моделей доступності SQL Database підтримує локальне та зональне резервування. Локальне резервування забезпечує відмовостійкість в межах центру обробки даних, в той час як зональне резервування ще більше підвищує відмовостійкість, захищаючи від збоїв зони доступності в регіоні. Проте зональне резервування не підтримуються стандартним рівнем послуг (DTU).

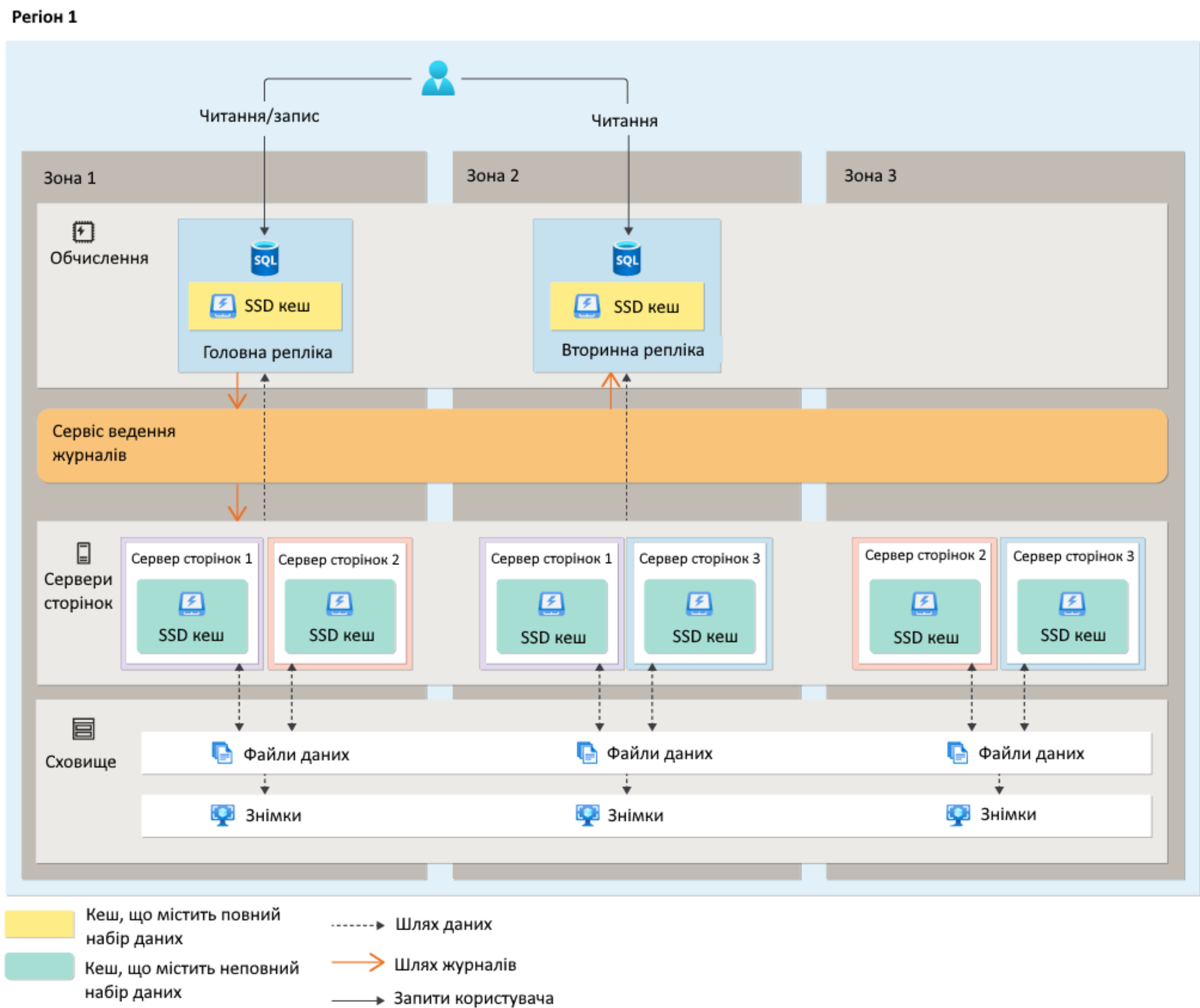

Рис. 2.2 – Гіпермасштабна модель Azure SQL

Azure SQL підтримує безсерверний рівень обчислень для окремих баз даних, який автоматично масштабує обчислення залежно від робочого навантаження і виставляє рахунки за обсяг обчислень, що використовуються за секунду. Рівень безсерверних обчислень також автоматично призупиняє роботу баз даних під час періодів бездіяльності, коли плата знімається лише за сховище, і автоматично відновлює роботу баз даних, коли активність відновлюється.

Налаштування продуктивності відбувається через вказання мінімального та максимального значень vCores. Ліміти пам'яті та вводу-виводу пропорційні вказаному діапазону vCore. Також підтримується

автоматичне призупинення, що налаштовується через визначення періоду часу, протягом якого база даних повинна бути неактивною, перш ніж вона буде автоматично призупинена. База даних автоматично відновлюється при наступному вході або іншій активності.

Вартість безсерверної бази даних складається з вартості обчислень і вартості зберігання. Вартість обчислень базується на кількості vCore і пам'яті, що використовується; коли навантаження нижче мінімальних налаштованих лімітів, вартість обчислень базується на вказаних мінімальній кількості vCores та мінімальній пам'яті. Коли база даних призупинена, плата за обчислення не нараховується, і присутні лише витрати на зберігання.

Безсерверні бази даних запускаються на машині з достатньою потужністю, щоб задовольнити попит на ресурси без перерви для будь-якого обсягу запитуваних обчислень в межах, встановлених максимальним значенням vCores. Якщо машина не може задовольнити попит на ресурси протягом декількох хвилин, відбувається автоматичне балансування навантаження. Під час балансування навантаження база даних залишається онлайн, за винятком короткого періоду у кінці операції, коли з'єднання розриваються.

2.3 Google Cloud

Google Cloud відрізняється зосередженістю на інтелектуальних рішеннях, що ґрунтуються на великих об’ємах даних. Платформа пропонує такі сервіси, як BigQuery для зберігання даних, TensorFlow у Google Cloud для машинного навчання та Vertex AI для управління штучним інтелеком.

Основні компоненти інфраструктури включають Compute Engine, який надає екземпляри віртуальних машин, та Google Kubernetes Engine (GKE) – керовану систему оркестрування, що підтримує контейнерні робочі навантаження на базі Kubernetes. Для зберігання об'єктів Google пропонує масштабовану систему з декількома класами - від стандартного до архівного рівня. Реляційні дані обслуговуються за допомогою Cloud SQL та Cloud Spanner. Для нереляційних даних пропонуються Bigtable та Firestore, що оптимізовані для горизонтально масштабованих робочих навантажень. BigQuery дозволяє зберігати петабайти даних і виконувати до них SQL-запити з мінімальною затримкою. Dataflow і Dataproc підтримують великомасштабну обробку даних у потокових і пакетних конвеєрах, інтегруючись з екосистемами Apache Beam, Spark і Hadoop. Vertex AI пропонує універсальний інтерфейс для навчання та розгортання моделей машинного навчання, використовуючи готові алгоритми, кастомні моделі TensorFlow або спеціалізовані апаратні прискорювачі. Ці можливості та мережа з глобальним балансуванням навантаження і низькою затримкою дозволяють створювати стабільні та найдійні розподілені системи.

Сервіси інфраструктури Google Cloud доступні в Північній Америці, Південній Америці, Європі, Азії, на Близькому Сході та в Австралії. Ці локації поділяються на регіони та зони. Регіони – це незалежні географічні області, які складаються із зон. Зони та регіони є логічними абстракціями фізичних ресурсів, що лежать в їх основі. Регіон складається з трьох або більше зон, розміщених у трьох або більше фізичних центрах обробки даних (табл 2.8).

Зона – це область розгортання ресурсів Google Cloud у регіоні. Зони представляються як окремий ресурс у регіоні. Для забезпечення відмовостійкі та захисту від несподіваних збоїв, додатки рекомендується розгортати в декількох зонах одночасно.

Таблиця 2.8 –Регіони Google Cloud

Ідентифікатор	Місцезнаходження
asia-east1	Тайвань
asia-northeast1	Токіо
asia-northeast2	Осака
europe-north1	Фінляндія
europe-southwest1	Мадрид
europe-west1	Бельгія
europe-west4	Нідерланди
europe-west8	Мілан
europe-west9	Париж
me-west1	Тель-Авів
us-central1	Айова
us-east1	Південна Кароліна
us-east4	Північна Вірджинія
us-east5	Колумбус
us-south1	Даллас
us-west1	Орегон

Google Cloud орієнтована на організації, які надають перевагу розширеній аналітиці даних, машинному навчанню та інтеграції з технологіями з відкритим вихідним кодом. Вона підходить для технологічних компаній, які можуть використовати BigQuery для великомасштабної аналітики або TensorFlow для розробки штучного інтелекту. Платформа Google Cloud також може бути привабливою для таких галузей, як роздрібна торгівля та логістика, завдяки спеціалізованим сервісам, як Vertex AI для оптимізації ланцюгів поставок, та стартапів з робочими навантаженнями Kubernetes через її лідерство в контейнеризації. Інтеграція з Google Workspace також може бути вирішальним фактором при виборі цієї хмарної платформи.

2.3.1 Cloud Run functions

Cloud Run functions – це безсерверне середовище виконання для створення та підключення хмарних сервісів [56]. Cloud Run functions прив'язуються до подій, що генеруються хмарною інфраструктурою та сервісами. Код виконується в повністю керованому середовищі Cloud Run, що дозволяє уникнути підтримки інфраструктури або управління серверами.

Розробником пропонуються використовувати ці функції для наступних цілей:

- обробка даних;
- веб-хуки;
- API;
- мобільні бекенди;
- IoT (Internet of things / Інтернет речей).

Існує дві версії функцій Cloud Run functions:

- Cloud Run functions (2-го покоління), які розгортають функції як сервіси в Cloud Run;
- Cloud Run functions (1-го покоління), раніше відомі як Cloud Functions, оригінальна версія функцій з обмеженими тригерами подій і конфігурацєю.

Порівняння особливостей версій можна побачити у таблиці 2.7.

Функції Cloud Run працюють у повністю керованому безсерверному середовищі, де Google керує інфраструктурою, операційними системами та середовищами виконання. Кожна функція працює у власному ізольованому контексті виконання, автоматично масштабується і має життєвий цикл, незалежний від інших функцій.

Функції Cloud Run підтримують декілька мовних середовищ виконання. Кожна з них містить стандартний набір системних пакетів, а також інструменти та бібліотеки, необхідні для цієї мови.

Google Cloud Run використовують повністю контейнеризовані робочі навантаження. Мовні середовища, а також версію функцій, що їми підтримуються, можна побачити у таблиці 2.9.

Таблиця 2.9 – Мовні середовища Cloud Run functions

Середовище	Версія Cloud Run functions	Ідентифікатор середовища
Node.js 22	Run functions	nodejs22
Node.js 20	1st gen, Run functions	nodejs20
Node.js 18	1st gen, Run functions	nodejs18
Node.js 16	1st gen, Run functions	nodejs16
Node.js 14	1st gen, Run functions	nodejs14
Node.js 12	1st gen, Run functions	nodejs12
Node.js 10	1st gen, Run functions	nodejs10
Python 3.12	1st gen, Run functions	python312
Python 3.11	1st gen, Run functions	python311
Python 3.10	1st gen, Run functions	python310
Python 3.9	1st gen, Run functions	python39
Python 3.8	1st gen, Run functions	python38
Python 3.7	1st gen, Run functions	python37
Go 1.22	Run functions	go122
Go 1.21	Run functions	go121
Go 1.20	Run functions	go120
Go 1.19	1st gen, Run functions	go119
Go 1.18	1st gen, Run functions	go118
Go 1.16	1st gen, Run functions	go116
Go 1.13	1st gen, Run functions	go113
Java 21	Run functions	java21
Java 17	1st gen, Run functions	java17
Java 11	1st gen, Run functions	java11
Ruby 3.3	1st gen, Run functions	ruby33
Ruby 3.2	1st gen, Run functions	ruby32
Ruby 3.0	1st gen, Run functions	ruby30
Ruby 2.7	1st gen, Run functions	ruby27
Ruby 2.6	1st gen, Run functions	ruby26
PHP 8.3	Run functions	php83
PHP 8.2	1st gen, Run functions	php82
PHP 8.1	1st gen, Run functions	php81
PHP 7.4	1st gen, Run functions	php74
.NET Core 8	Run functions	dotnet8
.NET Core 6	1st gen, Run functions	dotnet6
.NET Core 3	1st gen, Run functions	dotnet3

Окрім вищезазначених середовищ, підтримується будь-яка мова або фреймворк, якщо він працює у контейнері Linux і може використовувати HTTP-порт; контейнери на базі Windows не підтримують.

Cloud Run functions реалізують безсерверну парадигму; після розгортання функції автоматично керуються та масштабуються.

Cloud Run functions обробляють вхідні запити, призначаючи їх екземплярам функції. Залежно від обсягу запитів, а також кількості існуючих екземплярів функції, Cloud Run functions можуть призначити запит на існуючий екземпляр або створити новий. У випадках, коли обсяг вхідних запитів перевищує кількість існуючих екземплярів, функції Cloud Run можуть запустити кілька нових екземплярів для обробки запитів; надається можливість налаштувати максимальну кількість екземплярів, які можуть співіснувати в будь-який момент часу для певної функції.

Функції Cloud Run підтримують обробку декількох паралельних запитів на одному екземплярі функції для запобігання холодного запуску; вже теплий екземпляр може обробляти кілька запитів одночасно, проте функції Cloud Run не забезпечують ізоляцію між паралельними запитами, що обробляються одним і тим самим екземпляром функції.

Кожна функція має певний обсяг пам'яті, виділений для її використання. Обраний об'єм виділеної пам'яті відповідає певній потужності виділеного процесора.

Пам’ять обирається від 128 МіБ до 32 ГБ, що відповідає 0.83 vCPU та 8 vCPU відповідно. Проте функції можна налаштовувати із вказаною пам'яттю і vCPU окремо. При зміні обсягу пам'яті для функції, кількість vCPU буде перераховано відповідно, якщо вона не вказана окремо.

Середовище виконання функції включає файлову систему в пам'яті, яка містить файли і каталоги, розгорнуті разом з функцією. Каталог, що містить файли функції, доступний лише для читання, але решта файлової системи доступна для запису та її використання зараховується до використання пам'яті

функції. Кожна розгорнута функція ізольована від усіх інших функцій – навіть тих, що розгорнуті з того самого файлу. Вони не мають спільної пам'яті, глобальних змінних, файлових систем, тощо.

Cloud Run functions накладають певні обмеження на використання ресурсів у залежності від версії (табл. 2.10), які охоплюють 3 сфери:

- обмеження ресурсів;
- часові обмеження;
- обмеження за пропускною здатністю.

Таблиця 2.10 – Обмеження Cloud Run functions

Функція	Cloud Run functions (1st gen)	Cloud Run functions
Кількість функцій	1000 на регіон	1000 на регіон за виключенням кількості розгорнутих сервісів Cloud Run
Таймаут запиту	До 9 хвилин на виклик	До 60 хвилин для функцій, що запускаються через HTTP; До 9 хвилин для функцій, що запускаються за подіями
Розмір екземпляра	До 8 ГБ оперативної пам'яті з 2 vCPU	До 16GiB оперативної пам'яті з 4 vCPU
Паралельність	1 одночасний запит на екземпляр функції	До 1000 одночасних запитів на екземпляр функції
Розділення трафіку	Не підтримується	Підтримується
CloudEvents	Лише у середовищах Ruby, .NET та PHP	Підтримується у всіх мовних версіях
Максимальний розмір нестисненого HTTP-запиту	10 МіБ на окремий виклик	32 МіБ на окремий виклик
Максимальний розмір нестисненої HTTP відповіді	10 МіБ за виклик	10 МіБ за виклик для потокових відповідей. 32 МіБ для не потокових відповідей
Виклики API (READ)	5000 за 100 секунд на проект	1200 за 60 секунд на регіон
Виклики API (WRITE)	80 за 100 секунд на проект	60 за 60 секунд на регіон
Виклики API (CALL)	16 на 100 секунд на проект	Не застосовно

Існує два типи функцій Cloud Run:

- HTTP функції, які обробляють HTTP-запити і використовують HTTP тригери;
- функції, керовані подіями, які обробляють події з хмарного середовища і використовують тригери подій.

Хмарні функції, що викликаються по HTTP, швидко масштабуються для обробки трафіку, тоді як фонові функції масштабуються більш поступово та мають додаткові обмеження. Коли функція споживає весь виділений ресурс, цей ресурс стає недоступним доки ліміт не буде оновлено або збільшено.

Таким чином, Google Cloud Run надає повністю керовану платформу для розгортання та запуску контейнерних додатків або функцій у масштабованому середовищі, керованому подіями.

2.3.2 Spanner

Spanner – це повністю керована служба баз даних, яка предоставляється компанією Google. Вона пропонує узгодженість транзакцій у глобальному масштабі, автоматичну синхронну реплікацію для високої доступності та підтримку двох діалектів SQL: GoogleSQL (ANSI 2011 з розширеннями) та PostgreSQL [57].

Spanner відокремлює обчислювальні ресурси від сховища даних, що дозволяє легко масштабувати обчислювальні ресурси. Кожна додаткова обчислювальна потужність може обробляти як читання, так і запис, забезпечуючи легке горизонтальне масштабування. Spanner оптимізує продуктивність, автоматично керуючи шардінгом, реплікацією та обробкою транзакцій.

Spanner підтримує два діалекти SQL: GoogleSQL та PostgreSQL інтерфейс. PostgreSQL інтерфейс підтримує часткову сумісність з синтаксисом мови PostgreSQL з додатками для підтримки функцій Spanner, таких як таблиці з переплетенням і підказки. Spanner не підтримує деякі можливості PostgreSQL, зокрема наступні:

- тригери;
- SERIAL (тип даних);
- Дрібнозернистий контроль паралелізму;
- SAVEPOINT;
- транзакційний DDL;
- часткові індекси;
- розширення;
- користувацькі типи даних, функції та оператори.

Spanner підтримує два типи конфігурацій:

- регіональна – у регіональних екземплярах всі репліки бази даних знаходяться в одному географічному регіоні;

- багаторегіональна – у багаторегіональних екземплярах база даних реплікується на декілька регіонів для стійкості проти збоїв у окремих регіонах, масштабування, або зменшення затримки читання для користувачів, що знаходяться у різних регіонах.

У регіональній конфігурації Spanner створює 3 репліки даних і розподілює їх між 3 доменами, які називаються зонами, у межах одного географічного регіону. Кожна репліка містить повну копію бази даних і може обслуговувати запити на читання та запис. Для забезпечення узгодженості даних, Spanner використовує синхронну схему реплікації на основі Paxos, в якій репліки підтверджують кожен запит на запис. Запис фіксується, коли більшість реплік (кворум), погоджується на фіксацію запису. Для регіональних екземплярів 2 з 3 реплік повинні погодитися на фіксацію запису. Регіональні екземпляри гарантують 99,99% доступності.

У багаторегіональних конфігураціях Spanner створює від 5 реплік даних і розподілює їх між 3 (або більше) регіонами. Два регіони мають по дві репліки даних для читання і запису, розподілені між двома зонами, а третій регіон має один спеціальний тип репліки, який називається реплікою-свідком. Репліки-свідки не зберігають повну копію баз даних, проте вони беруть участь у голосуванні за внесення змін до бази даних. Оскільки репліки-свідки не зберігають повну копію даних, вони не можуть обслуговувати запити на читання. Мультирегіональні екземпляри надають вищі гарантії доступності (99,999%) та глобальний масштаб, однак, це досягається за рахунок більших затримок при записі.

Spanner ділить дані на частини, які називаються «сплітами». Кожен спліт має декілька копій, або реплік. Щоб забезпечити надійну узгодженість, одна з цих реплік обирається (за допомогою Paxos) в якості лідера для спліту. Репліка-лідер для розділеного голосування відповідає за обробку записів. Всі запити клієнтів на запис спочатку надходять до лідера, який реєструє запис, а потім надсилає його іншим реплікам з правом голосу. Після того, як кожна

репліка завершує запис, вона відповідає лідеру і голосує за те, чи повинен лідер зафіксувати цей запис. Коли більшість голосуючих реплік голосують за збереження запису, лідер вважає, що запис збережено. У багаторегіональних екземплярах спліти реплікуються на кілька регіонів та підтримується вказання регіону лідера (за замовчуванням). Якщо вказаний регіон лідеру виходить з ладу або стає недоступним, Spanner переміщує лідерів в інший регіон, а коли цей регіон лідера знову стає доступним, лідер автоматично переміщується назад у вказаний регіон. Багаторегіональні конфігурації Spanner також підтримують репліки тільки для читання. Розбиття на спліти відбувається автоматично, коли Spanner виявляє високе навантаження на читання або запис.

Гарантією доступності Spanner є розподілена файлова система Google, Colossus. Використання Colossus дозволяє відокремити сховище файлів від сервісу баз даних. Spanner має архітектуру «нічого спільного», і оскільки будь-який сервер у кластері може читати з Colossus, репліки можуть швидко відновлюватися після збоїв. Репліки читання-запису Spanner передають дані до Colossus, де вони реплікуються 3 рази. Враховуючи, що кластер Spanner має три репліки читання-запису, дані реплікуються 9 разів.

Spanner пропонує три версії, які надають різні можливості за різними ціновими категоріями (табл. 2.11):

- Standard – надає повний набір встановлених можливостей, які включають всі функції, що були загальнодоступними до 24 вересня 2024 року, а також деякі додаткові можливості, такі як резервне копіювання за розкладом, в регіональних конфігураціях екземплярів;
- Enterprise – базується на версії Standard і пропонує багатомодельні можливості, включаючи Spanner Graph, а також повнотекстовий пошук і векторний пошук. Окрім цього, пропонується спрощена експлуатація та захист даних за допомогою керованого автоматичного масштабування та інкрементного резервного копіювання;

- Enterprise Plus – призначена для робочих навантажень, які потребують 99,999% доступності, з конфігураціями для кількох регіонів і підтримкою георозділів.

Таблиця 2.11 – Версії Spanner

Функціонал / Версія	Standard	Enterprise	Enterprise Plus
Доступність	99.99%	99.99%	До 99.999%
Конфігурації	Регіональна	Регіональна. Опціональні репліки для читання	Регіональна, дворегіональна, багаторегіональна. Опціональні репліки для читання
Багатомодельні можливості	Реляційна. Ключ-значення.	Реляційна. Ключ-значення. Spanner Graph	Реляційна. Ключ-значення. Граф
Можливості пошуку	-	Повнотекстовий пошук. Векторний пошук	Повнотекстовий пошук. Векторний пошук
Управління ресурсами	Утиліта автоматичного масштабування	Утиліта автоматичного масштабування. Керований автоматичний масштабувач	Утиліта автоматичного масштабування. Керований автоматичний масштабувач. Географічне розбиття
Захист даних	Стандартні резервні копії. 7 денне відновлення до певного моменту. Резервне копіювання за розкладом.	Стандартні резервні копії. 7 денне відновлення до певного моменту. Резервне копіювання за розкладом. Інкрементні резервні копії.	Стандартні резервні копії. 7 денне відновлення до певного моменту. Резервне копіювання за розкладом. Інкрементні резервні копії.

При використанні Spanner спочатку необхідно створити екземпляр, для якого обирається версія, конфігурація та обчислювальна потужність. Обчислювальна потужність визначає кількість ресурсів сервера та сховища,

доступних для баз даних в екземплярі. При створенні екземпляру необхідно вказати його обчислювальну потужність як кількість процесорних одиниць (кратну 100) або як кількість вузлів, при цьому 1000 процесорних одиниць дорівнюють 1 вузлу. Spanner не має режиму очікування, обчислювальна потужність Spanner є виділеним ресурсом, і навіть при відсутності навантаження, Spanner виконує фонові роботи з оптимізації та захисту даних. Збільшення обчислювальної потужності не призводить до збільшення кількості реплік (які є фіксованими для даної конфігурації екземпляра), а лише збільшує ресурси кожної репліки в екземплярі. Один екземпляр підтримує до 100 БД.

Версії Enterprise та Enterprise Plus підтримують використання автоматичного масштабувача. При використанні цієї утиліти, Spanner автоматично масштабує розмір екземпляра, реагуючи на зміни у робочому навантаженні або потребах на сховище екземпляра. Масштабувач визначає, скільки обчислювальних потужностей потрібно, виходячи з наступних параметрів:

- цільове навантаження на процесор;
- цільове використання сховища;
- мінімальна налаштована межа;
- максимальна налаштована межа.

Кожен вимір масштабування генерує рекомендований розмір екземпляра, і Spanner автоматично використовує найбільший з них. Зі зміною обсягу обчислювальних потужностей Spanner постійно оптимізує сховище та перерозподіляє дані між усіма серверами, для рівномірного розполення трафіку і уникання перевантаження окремих серверів.

2.4 Висновки по розділу

AWS Lambda підтримує певний набір мов (Node.js, Python, Java, Go, Ruby, .NET), а також пропонує користувацькі середовища виконання (табл. 2.3), проте всі середовища базуються на Linux (Amazon Linux) без можливості виконання функцій на базі Windows. Lambda вимагає певний формат пакування (ZIP-архів або образ контейнера) і не дозволяє створювати довільне середовище операційної системи.

Azure Functions, у свою чергу, підтримує широкий спектр мов (C#, JavaScript/Node.js, Python, Java, PowerShell та користувацькі обробники) і може працювати як на хостах Linux, так і на хостах Windows, залежно від обраного тарифного плану.

Google Cloud Run використовує інший підхід, виконуючи повністю контейнеризовані робочі навантаження. Підтримується будь-яка мова або фреймворк, якщо він працює у контейнері Linux і може використовувати HTTP-порт, що забезпечує гнучкість виконання. Cloud Run не підтримує контейнери на базі Windows.

З точки зору обмежень на ресурси, AWS Lambda дозволяє виділяти функціям від 128 МіБ до 10240 МіБ пам'яті та пропорційну процесорну потужність та підтримують максимальний час виконання у 15 хвилин. Максимальна кількість одночасних функцій становить за замовчуванням 1000 на регіон, проте може бути збільшена за запитом.

Azure Functions виділяють до 1,5 ГБ пам'яті (Consumption plan) на екземпляр з пропорційним розподілом процесорних ресурсів, а час виконання за замовчуванням обмежений 10 хвилинами. Дорожчі плани надають більше пам'яті та можливість працювати довше. Паралельність на екземпляр у Functions обмежується у залежності від обраного плану (600 для Consumption); кожен екземпляр зазвичай обробляє кілька паралельних запитів залежно від часу виконання мови та тригерів.

Google Cloud Run дозволяє користувачам налаштовувати процесор (до 8 vCPU) і пам'ять (до 16 ГБ) для кожного контейнера. Паралельність для кожного екземпляра налаштовується до 1 000 одночасних запитів. Таймаути запитів на Cloud Run можуть досягати 60 хвилин. Cloud Run, по суті, надає розробникам налаштовуване мікросередовище на основі міні-контейнерів.

Lambda та Azure функції запускаються подіями з інших сервісів або через API-шлюзи. Cloud Run відповідає на HTTP(S) запити безпосередньо, не потребуючи окремого шлюзу, хоча його можна інтегрувати з іншими сервісами обробки подій Google Cloud. Крім того, важливою є інтеграція з відповідними екосистемами: Lambda з сервісами AWS, такими як S3 та DynamoDB; Azure Functions з Azure Storage або Event Hubs; Cloud Run можна поєднати з Pub/Sub або Eventarc.

Таким чином, AWS Lambda, Azure Functions та Google Cloud Run помітно відрізняються за підтримуваними середовищами виконання, гнучкістю ОС, охопленням регіонів та конфігурацією ресурсів. Lambda обмежена операційними системами Amazon Linux і співвідношеннями пам'яті/процесора, має зрозумілу модель ціноутворення і доступна в багатьох регіонах. Azure Functions пропонує 2 ОС (Windows або Linux), різноманітність цінових моделей і хостингу, а також найбільшу глобальну гнучкість з більш розширеними варіантами конфігурації. Cloud Run орієнтована на контейнери, та тарифікується за використання процесора та пам'яті під час запитів, але вимагає управління контейнерами і базується виключно на Linux. Всі платформи мають проблему холодного старту, дозволяють швидко масштабуватися і мають певні обмеження на ресурси та середовища, які необхідно детально розглядати перед вибором платформи для окремого завдання.

Amazon Aurora, Azure SQL Database та Google Cloud Spanner є керованими системами реляційних баз даних, однак їхні архітектури, підходи до масштабування, моделі узгодженості та підтримка SQL діалектів суттєво

відрізняються, що впливає на вартість, складність та придатність для різних навантажень.

Amazon Aurora пропонує сумісність з MySQL та PostgreSQL. Вона зберігає дані в розподіленому сховищі, яке автоматично масштабується до 128 ТБ. Потужності обчислення і зберігання масштабуються незалежно, що дозволяє створювати більшу кількість реплік для читання або масштабувати екземпляри. Aurora включає в себе «безсерверну» конфігурацію (Aurora Serverless v2), що дозволяє змінювати ємність малими інкрементами та дозволяє постійно масштабувати потужності у залежності від навантаження. Aurora забезпечує високу узгодженість читання після запису і підтримує до 15 реплік читання. Завдяки підтримці MySQL або PostgreSQL багато додатків можна мігрувати до цієї системи з мінімальними змінами коду. Проте Aurora залишається регіонально прив'язаною за замовчуванням, хоча дані можна копіювати в декілька регіонів для читання та аварійного відновлення, вона не забезпечує повноцінний запис в декілька регіонів з єдиною логічною базою даних. Іншим потенційним недоліком є складність ціноутворення Aurora з окремою оплатою за обчислення, зберігання та запити. І хоча у теорії Aurora Serverless може зменшувати витрати при відсутності навантаження, холодний запуск є занадто повільнім для більшості навантажень, а вартість підтримки теплого кластеру у декілька разів перевищує вартість аналогічної БД з подібною ємністю.

Azure SQL Database підтримує T-SQL і забезпечує сумісність з SQL Server. Вона орієнтована на різні варіанти розгортання: окремі бази даних, еластичні пули та рівень Hyperscale для великомасштабних робочих навантажень. Hyperscale може масштабувати сховище до 128 ТБ також і розділяє потужності обчислень і зберігання. Він пропонує швидке вертикальне масштабування обчислювальних вузлів і сховища. Архітектура Hyperscale використовує сервери сторінок і репліки для розподілу вводу/виводу зі сховища, що забезпечує вищу пропускну здатність і меншу затримку у

великих базах даних. Однак, горизонтальне масштабування записів на кілька кінцевих точок, доступних для запису, не підтримується. Безсерверний рівень може, як і Aurora, призупиняти роботу бази даних, коли вона неактивна, знижуючи витрати, а також автоматично відновлювати і масштабувати процесор і пам'ять залежно від робочого навантаження. Azure SQL забезпечує надійну узгодженість у межах регіону та пропонує такі функції, як автоматичне налаштування та вбудована аналітика продуктивності, проте сценарії запису з кількох регіонів вимагають ручного шардингу або використання інших служб Azure.

Google Cloud Spanner використовує власну архітектуру і TrueTime API для підтримки глобально узгодженого впорядкування транзакцій на основі міток часу. Вона не є традиційною базою даних SQL, та була створена з самого початку для горизонтального масштабування по регіонах, забезпечуючи ACID транзакції. На відміну від Aurora або Azure SQL, потужності сховища та обчислення Spanner масштабуються автоматично при додаванні вузлів. Вона може працювати з базами даних, які містять петабайти даних і тисячі операцій запису в секунду в різних географічних регіонах, без ручного шардингу. Однак, Spanner застосовує специфічні схеми, і хоча вона використовує діалект SQL, він не є повністю сумісним з синтаксисом MySQL, PostgreSQL або T-SQL. Перенесення існуючих додатків на Spanner може вимагати значних змін у коді та коригування схеми БД. На відміну від безсерверного варіанту Aurora або Azure Hyperscale, Spanner не масштабується до нуля: плата за виділені ресурси знімається незалежно від навантаження на БД. З точки зору узгодженості, Spanner пропонує сильну глобальну узгодженість, проте вартість використання цієї системи може бути високою і визначення її складнішим, враховуючи кількість процесорних одиниць, використання сховища та міжрегіональну реплікацію. Окрім цього, Spanner доступний лише в Google Cloud і працює на пропрієтарній інфраструктурі, що знижує його портативність.

Таким чином, Aurora і Azure SQL більше покладаються на вертикальне масштабування і репліки читання, а також відділення потужностей обчислення від сховища. Aurora та Azure SQL підтримують MySQL, PostgreSQL та T-SQL відповідно, але реплікація між регіонами потребує використання додаткових сервісів. Spanner розподіляє дані між вузлами та регіонами без ручного втручання у залежності від налаштованих лімітів. Spanner підходить для глобально розподілених робочих навантажень з інтенсивним записом, які потребують сильної узгодженості, але вимагає відходу від звичайних механізмів баз даних, адаптації до своєї моделі, більш високих базових витрат і деякої втрати сумісності зі звичними SQL-движками. Додаткові ліміти варто розглядувати для окремих робочих навантажень.

РОЗДІЛ 3. ОЦІНКА ПРИДАТНОСТІ БЕЗСЕРВЕРНИХ РІШЕНЬ ДЛЯ МАСШТАБОВАНИХ СЕРВІСІВ НА ПРИКЛАДІ AWS

Цей розділ присвячений опису експерименту, спрямованому на оцінку придатності AWS Lambda та Amazon Aurora Serverless для розгортання масштабованих сервісів.

Вибір платформи AWS, як основи для цього експерименту, зумовлений її зрілістю та широким використанням, що дозволить використовувати результати у якості точки відліку, з якою можна порівнювати інші платформи. Тісна інтеграція цих сервісів дає можливість дослідити рішення, що складається з безсерверних системи та бази даних у контрольованих умовах.

Методологія, що застосована в цих експериментах, слідує структурованому, інкрементальному підходу, покликаному прояснити, як додаткові операційні шари та масштабування впливають на час виконання функції на платформі AWS. Спочатку було реалізовано базову Lambda функцію з використанням хешування Argon2. Цей етап був направлений на оцінку ресурсоємних операцій. На другому етапі функція була розширена для отримання секретних параметрів зі сховища параметрів AWS Systems Manager (SSM), що додало мережеву затримку та аутентифікацію з зовнішнім (з точку зори віртуальної мережі) сервісом. Після цього функцію було розширено, додаваючи використання бази даних Aurora Serverless v2. Цей етап дозволив дослідити підключення до бази даних та виконання запиту, імітуючи реалістичний потік даних. На четвертому етапі було додано генерацію JSON веб-токену (JWT), що відображає зазвичай необхідний крок для управління сеансами. На останньому етапі було виконано ряд одночасних (з мінімальною затримкою) викликів функції для порівняння часу виконання окремої функції з багатьма, дослідити масштабування Aurora Serverless, та визначити придатність описаної платформи для використання у масштабованих сервісах.

3.1 Аналіз впливу виділених потужностей Lambda на час виконання функції

Експериментальні умови були стандартизовані наступним чином: середовище виконання AWS Lambda Node.js 20.x на архітектурі x86-64.

Було протестовано п'ять різних конфігурацій пам'яті – 128, 256, 512, 1024 та 2048 МіБ - для того, щоб визначити вплив обраних обчислювальних ресурсів на швидкість виконання.

Для кожного етапу та кожної конфігурації було виконано 30 холодних і 30 теплих запусків, загалом 60 вимірювань для кожного значення ПДД на етап. Вимірювання холодного запуску включають в себе накладні витрати на ініціалізацію функцій, завантаження коду та налаштування середовища виконання. Вимірювання теплого запуску уникають ці затримки і перевіряють швидкість виконання самої функції.

Назви етапів відображають доданий функціонал та фігурують у таблицях наступним чином:

- етап 1 – Argon2;
- етап 2 – SSM;
- етап 3 – Aurora;
- етап 4 – JWT.

На початковому етапі експерименту метою було охарактеризувати продуктивність і вартість обчислювально інтенсивної операції – хешування паролів за алгоритмом Argon2 [58] при використанні функції Lambda з різними конфігураціями пам'яті.

Пакет Argon2 використовувався з наступними параметрами:

- hashLength – 32;
- timeCost – 3;
- memoryCost – 65536;
- parallelism – 4;

- type – argon2id.

При найменшій кількості виділеної пам'яті (128 МіБ) тривалість виконання Argon2 була очікувано найвищою, і становила в середньому 4064,47 мс для холодних запусків (табл. 3.1) і 3185,10 мс для теплих (табл. 3.2).

Таблиця 3.1 – Середній час виконання холодної Lambda функції

ПДД, МіБ	Час, мс			
	Argon2	SSM	Aurora	JWT
128	4064,47	8036,04	8790,83	9518,49
256	2074,28	4007,14	4439,54	4778,19
512	1026,10	1967,15	2146,18	2295,47
1024	503,27	970,15	1095,66	1168,74
2048	260,35	540,81	658,69	709,64

Зі збільшенням обсягу пам'яті час роботи Argon2 зменшувався пропорційно, включаючи найвищу конфігурацію у 2048 МіБ (рис. 3.1 – 3.2).

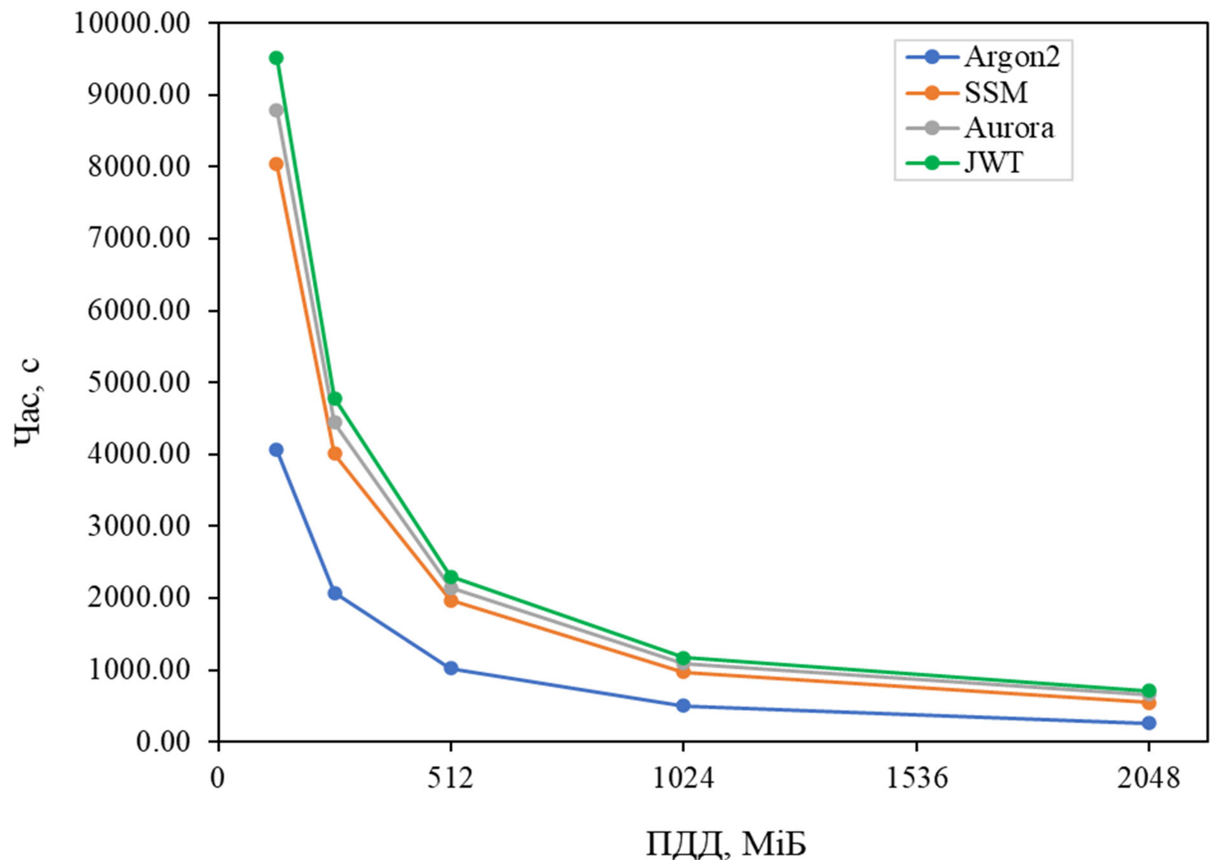

Рис. 3.1 – Графік залежності часу холодного старту Lambda від об’єму ПДД

Таблиця 3.2 – Середній час виконання теплої Lambda функції

ПДД, МіБ	Час, мс			
	Argon2	SSM	Aurora	JWT
128	3185,10	3363,09	3623,36	3519,04
256	1585,09	1610,78	1810,59	1826,59
512	756,31	822,84	908,69	926,12
1024	375,26	428,57	489,77	492,66
2048	183,46	233,53	297,44	288,93

Оскільки продуктивність Argon2 тісно пов'язана з доступністю процесора, спостережуване скорочення часу виконання є очікуваним і свідчить про те, що виділені ресурси масштабуються лінійно.

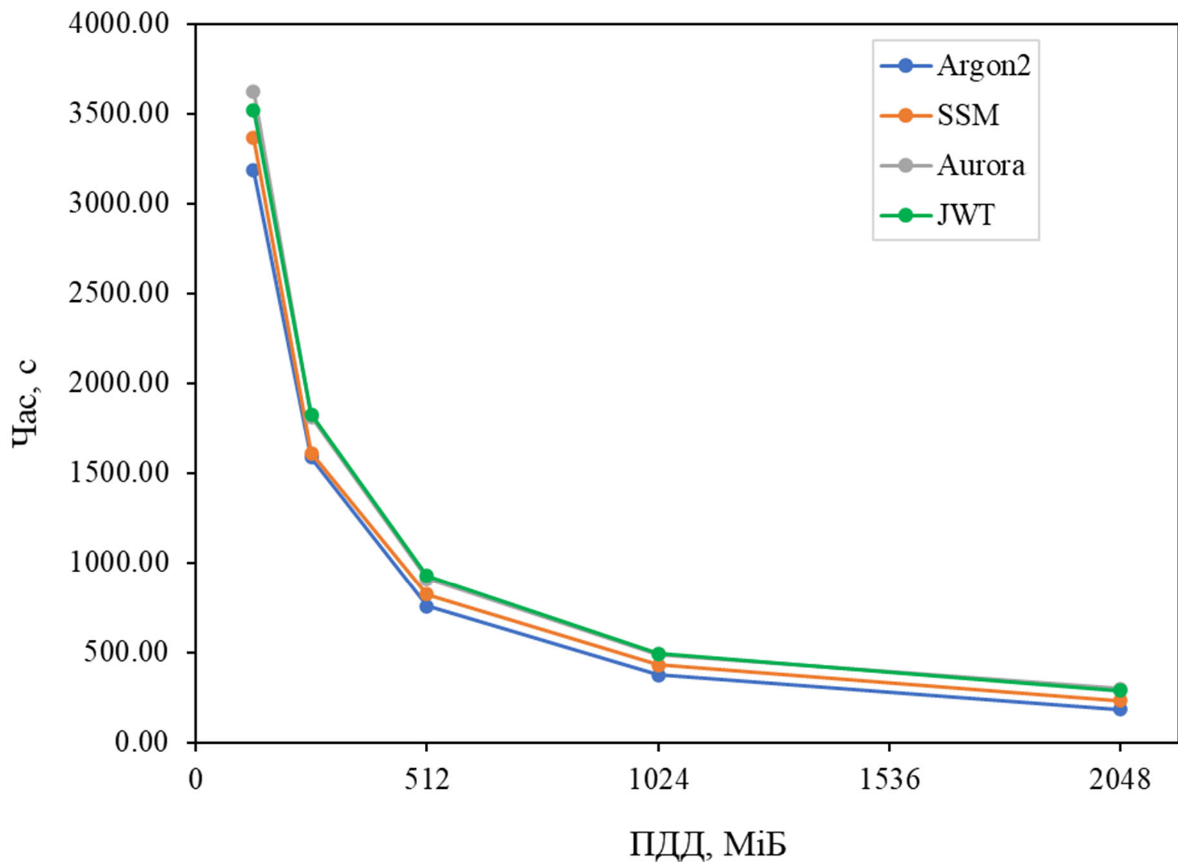

Рис. 3.2 – Графік залежності часу теплого старту Lambda від об’єму ПДД

Стандартні відхилення (σ) і коефіцієнти варіації (c_v) були обчислені для кількісної оцінки як абсолютної, так і відносної варіативності часу виконання функції (табл. 3.3 – 3.4).

Таблиця 3.3 – Показники варіативності часу виконання холодної Lambda

ПДД, МіБ	Argon2		SSM		Aurora		JWT	
	σ, мс	c_v, %	σ, мс	c_v, %	σ, мс	c_v, %	σ, мс	c_v, %
128	109,41	2,69	345,62	4,30	279,20	3,18	277,50	2,92
256	43,37	2,09	101,28	2,53	149,72	3,37	197,96	4,14
512	28,98	2,82	45,31	2,30	120,54	5,62	133,29	5,81
1024	13,29	2,64	24,54	2,53	47,57	4,34	30,28	2,59
2048	14,46	5,55	21,56	3,99	51,07	7,75	48,22	6,79

Хоча, у залежності від конфігурації ці показники змінювалися, мінливість часу виконання функції залишалась слабкою (<=5,55%).

Таблиця 3.4 – Показники варіативності часу виконання теплої Lambda

ПДД, МіБ	Argon2		SSM		Aurora		JWT	
	σ, мс	c_v, %	σ, мс	c_v, %	σ, мс	c_v, %	σ, мс	c_v, %
128	65,04	2,04	59,68	1,77	121,99	3,37	158,46	4,50
256	32,89	2,07	48,80	3,03	67,30	3,72	101,34	5,55
512	21,00	2,78	25,69	3,12	31,54	3,47	18,19	1,96
1024	10,74	2,86	13,32	3,11	19,75	4,03	24,67	5,01
2048	4,53	2,47	9,18	3,93	14,77	4,97	20,37	7,05

Показники вартості (виражені в термінах вартості одного виклику) мало відрізнялися незалежно від обраної конфігурації. Для холодного запуску найдешевшою виявилась конфігурація з 1024 МіБ ПДД (табл. 3.5, рис. 3.3), проте різниця з другою найдешевшею конфігурацією склала лише 1,31%.

Таблиця 3.5 – Середня вартість виконання холодної Lambda функції

ПДД, МіБ	Кошт, USD			
	Argon2	SSM	Aurora	JWT
128	8,5354E-06	16,8757E-06	18,4607E-06	19,9888E-06
256	8,7120E-06	16,8300E-06	18,6461E-06	20,0684E-06
512	8,5166E-06	16,3274E-06	17,8133E-06	19,0524E-06
1024	8,4047E-06	16,2015E-06	18,2976E-06	19,5180E-06
2048	8,6697E-06	18,0088E-06	21,9344E-06	23,6310E-06

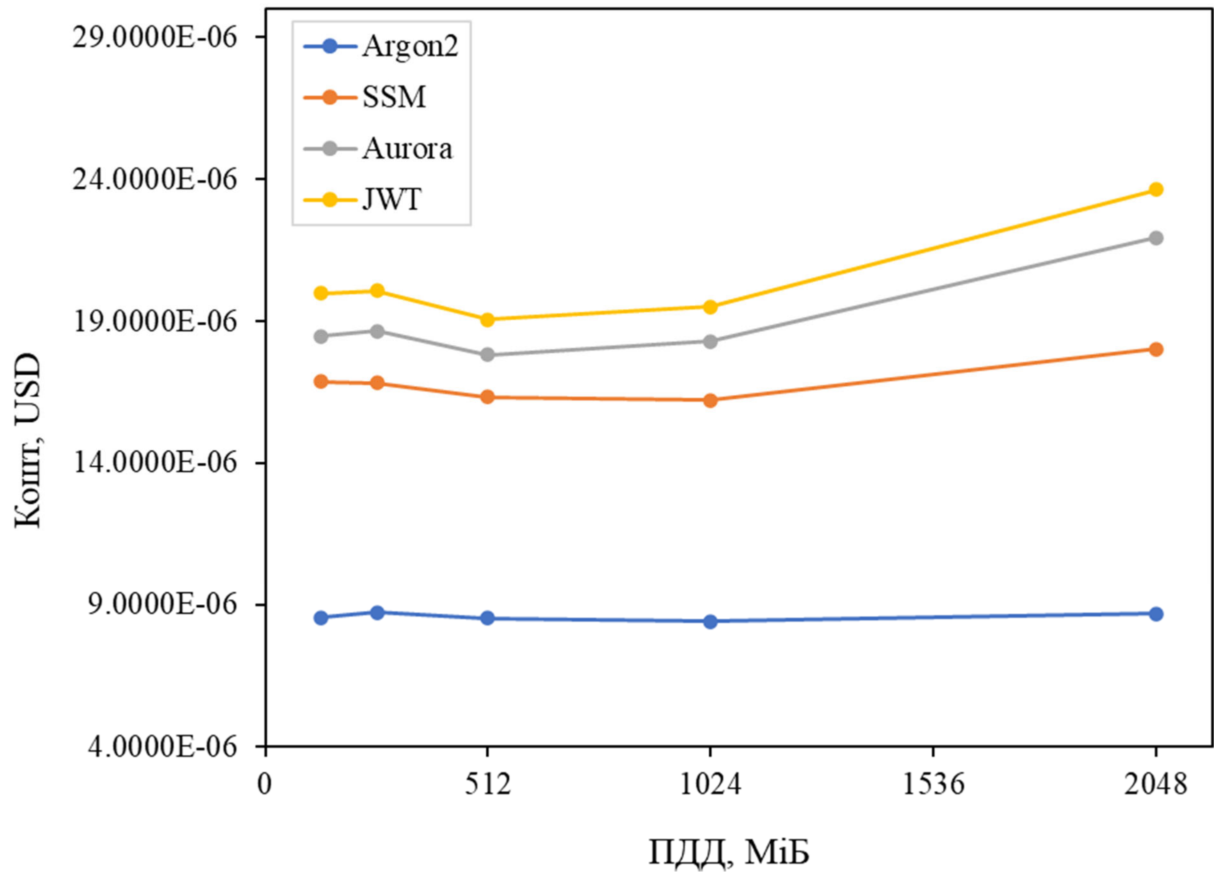

Рис. 3.3 – Графік залежності кошту виконання холодної Lambda від ПДД

Найдешевшою конфігурацією для теплого запуску виявилсь 2048 МіБ ПДД (табл. 3.6, рис. 3.4).

Таблиця 3.6 – Середня вартість виконання теплої Lambda функції

ПДД, МіБ	Кошт, USD			
	Argon2	SSM	Aurora	JWT
128	6,6887E-06	7,0625E-06	7,6090E-06	7,3900E-06
256	6,6574E-06	6,7653E-06	7,6045E-06	7,6717E-06
512	6,2773E-06	6,8295E-06	7,5422E-06	7,6868E-06
1024	6,2668E-06	7,1571E-06	8,1791E-06	8,2273E-06
2048	6,1093E-06	7,7765E-06	9,9049E-06	9,6212E-06

Таким чином при такому навантаженні, що складається виключно з обчислювальних операцій, найшвидша конфігурація може виявитись найдешевшею.

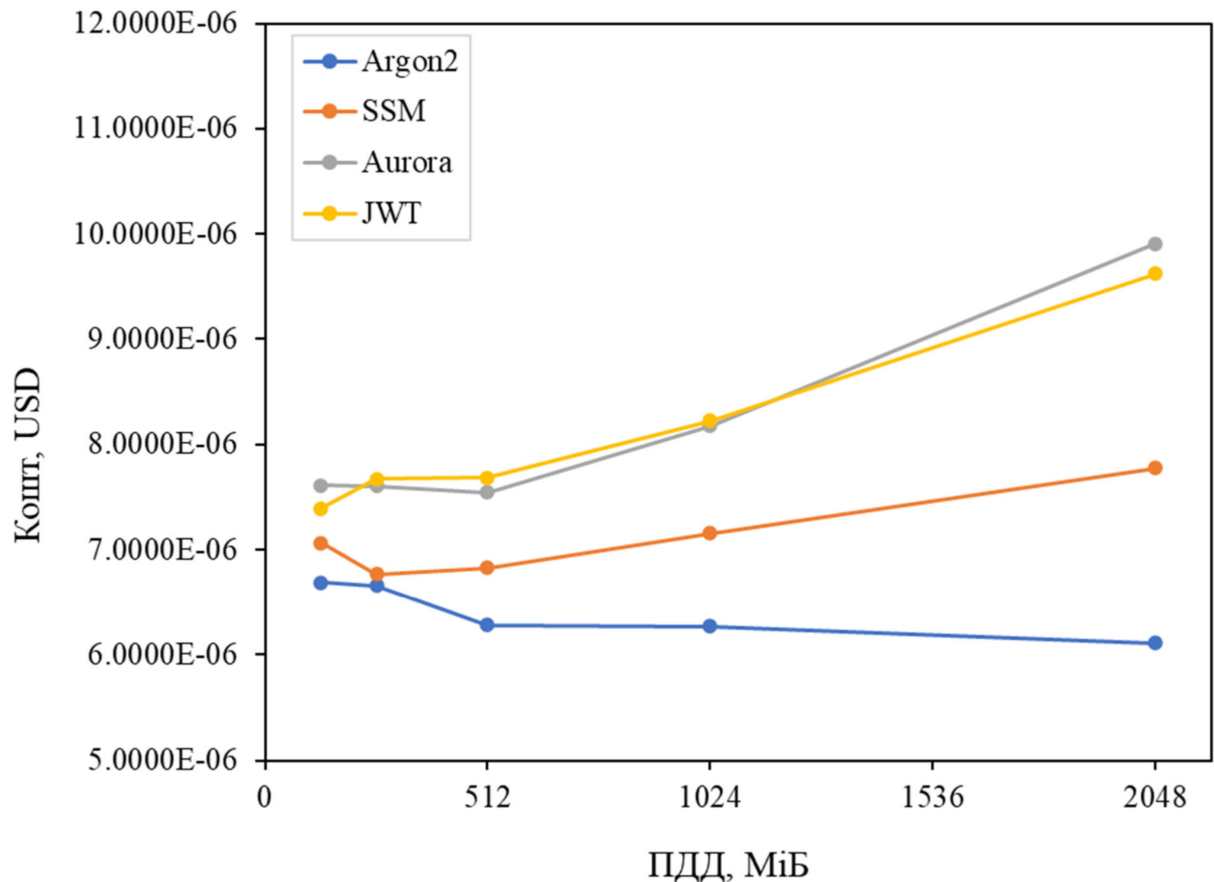

Рис. 3.4 – Графік залежності кошту виконання теплої Lambda від ПДД

На другому етапі експерименту складність функції було збільшено шляхом інтеграції операцій отримання даних з AWS Systems Manager (SSM) Parameter Store. Різниця полягала у виконанні додаткового запиту до SSM для отримання параметрів з'єднання з базою даних (ім'я бази даних, порту, ім'я користувача та паролю). Всі інші умови залишилися тими самими.

Впровадження використання SSM мало сильний вплив на швидкість виконання функції. При найменшому виділенні пам'яті (128 МіБ) тривалість холодного запуску становила близько 8036,04 мс, у порівнянні з минулим показником у 4064,47 мс. Таке фактичне подвоєння часу виконання показує вартість мережевих операцій та викликів AWS SDK при обмежених обчислювальних ресурсах. Масштабування пам'яті до 2048 МіБ зменшило тривалість холодного запуску до 540,81 мс, проте все ще підтримувало

пропорцію 1:2. Сповільнення теплої функції також відбулось, але не таке різке. Час виконання при 128 МіБ збільшився у середньому з 3185,10 мс до 3363,09 мс, а при 2048 МіБ – з 183,46 мс до 233,53 мс.

Показники варіативності суттєво не змінилися, а вартість за виконання показала подібні тенденції з 128 МіБ до 1024 МіБ, проте помітно виросла при переході до 2048 МіБ.

Ці результати показують, що додавання зовнішнього виклику до Parameter Store призводить до додаткових затримок , що залежать не лише від виділених функції ресурсів – окрім них сказуються мережеві затримки, передача ключів шифрування та операції вводу/виводу. В результаті, переваги масштабування пам'яті (і, відповідно, процесора), що спостерігались раніше, дещо зменшуються через фактори, які знаходиться поза прямим контролем користувача.

На третьому етапі було впроваджено взаємодію з базою даних шляхом включення запиту до екземпляру Aurora Serverless v2 (PostgreSQL). Таким чином функція:

- виконує хешування через Argon2,
- отримує деталі конфігурації з SSM,
- підключається до Aurora Serverless v2 для отримання збереженого хешу пароля, пов'язаного із запитуваним логіном.

Отримані дані відображають подальше очікуване збільшення затримок у порівнянні з минулим етапом. При найменшому об’ємі пам'яті (128 МіБ) тривалість холодного запуску досягла 8790,83 мс, додаючи приблизно 750 мс до останнього результату. Теплі старти на 128 МіБ мали схожу тенденцію, в середньому близько 3623,36 мс, порівняно з ~3363,09 мс для другого етапу.

Мінливість часу виконання функції дещо зросла, проте коефіцієнт варіації не перевищив 7,75%.

Вартість за виконання мало змінювалась на проміжку від 128 до 1024 МіБ, проте суттєво виросла (приблизно на 20%) при переході з конфігурації з 1024 до 2048 МіБ ПДД.

Таким чином, запити до бази даних додають ще один вимір складності, на який впливають як обмеження процесора, так і мережеві, разом з обмеженнями вводу/виводу. Масштабування процесора залишається ефективним, але час виконання додатково залежить від мережевих умов і внутрішніх механізмів масштабування Aurora. Aurora Serverless v2 пропонує автоматичне налаштування потужності, але ці налаштування можуть призвести до нестабільних результатів при масштабуванні.

На четвертому етапі експерименту до функції Lambda було додано генерацію JSON Web Token (JWT) і повернення його у файлі cookie.

Результати показують помітне збільшення часу холодного запуску функції (з 8790,83 до 9518,49 мс при 128 МіБ ПДД), але при цьому час теплого запуску суттєво не збільшився, а у деяких випадках навіть зменшився, що можна пояснити затримками SMM та Aurora, які перевищують затримку функції, викликану генерацією токена.

Мінливість часу виконання функції залишилась стабільно малою, та коефіцієнт варіації не перевищив 7,05%.

Вартість за виконання, як і очікувалось, підтримує тенденцію подібних витрат з 128 до 1024 МіБ та помітного зросту при переході з 1024 до 2048 МіБ у конфігурації Lambda.

Дані, отримані у результаті цього етапу показують, що найбільший вплив на час виконання надають завдання, пов'язані з процесором (Argon2), зверненням до зовнішніх сервісів (SSM) та операціями з базами даних (Aurora). Хоча внесок генерації JWT є помітним, він є відносно незначним у порівнянні з попередніми кроками.

3.2 Аналіз впливу масштабування Lambda та Aurora на час виконання функції

На цьому етапі експерименту фокус змістився з ізольованих виконань функції Lambda в контрольованих умовах на оцінку її поведінки під впливом десятків та сотень одночасних запитів. Метою цього етапу є оцінка того, як раніше спостережувані тенденції продуктивності ведуть себе, коли накладаються реальні сценарії навантаження. Кожна тестова конфігурація відповідала попередній методології: функції виконували повний робочий процес (хешування Argon2, отримання параметрів з SSM, запит до бази даних Aurora та генерація JWT), об’єм виділеної пам'яті було змінено (128, 256, 512, 1024 та 2048 МіБ), і виконувалось відокремлення холодних та теплих запусків. Додатково, збиралися дані щодо тривалості запиту до читача кластеру Aurora та виділення потужностей для оцінки масштабованості БД.

Зібрані дані холодних запусків (табл. 3.7) вказують на загальну тенденцію, що узгоджується з попередніми спостереженнями: збільшення виділенної ПДД Lambda значно зменшує середню загальну тривалість її виконання, проте відбувається додаткове збільшення середнього часу виконання функції (з 1,25 до 9,8%). Для теплих запусків час виконання зростає ще помітніше (з 6,17 до 18.08%), а коефіцієнт варіації досягає 22,41% (табл. 3.8).

Таблиця 3.7 – Результати одночасного запуску багатьох холодних Lambda

ДД, МіБ	Lambda				Запит до БД		
	Час, мс	σ, мс	c_v, %	Зрост часу виконання відносно одного запуску, %	Час, мс	σ, мс	c_v, %
128	9637,80	489,62	5,08	1,25	529,62	228,57	43,16
256	4986,88	411,05	8,24	4,37	502,57	292,62	58,23
512	2520,25	279,12	11,08	9,79	291,51	250,23	85,84
1024	1216,62	125,87	10,35	4,10	89,72	118,49	132,07
2048	736,31	77,83	10,57	3,76	81,60	72,43	88,77

Таким чином мінливість теплих запусків змінюється з слабкої до середньої під час помірних паралельних навантажень .

Таблиця 3.8 – Результати одночасного запуску багатьох теплих Lambda

ДД, МіБ	Lambda				Запит до БД		
	Час, мс	σ, мс	c_v, %	Зрост часу виконання відносно одного запуску, %	Час, мс	σ, мс	c_v, %
128	4143,53	546,75	13,20	17,75	210,56	93,46	44,39
256	2104,64	294,88	14,01	15,22	100,32	60,63	60,44
512	1093,56	245,08	22,41	18,08	199,76	229,57	114,92
1024	523,06	100,89	19,29	6,17	53,75	96,14	178,87
2048	319,41	49,43	15,47	10,55	37,70	45,96	121,92

Одночасно з AWS Lambda було проведено аналіз поведінки Amazon Aurora Serverless v2 та її реакції на швидкозмінний попит (рис. 3.5 – 3.6). Aurora в цих експериментах слугувала сховищем даних для функцій.

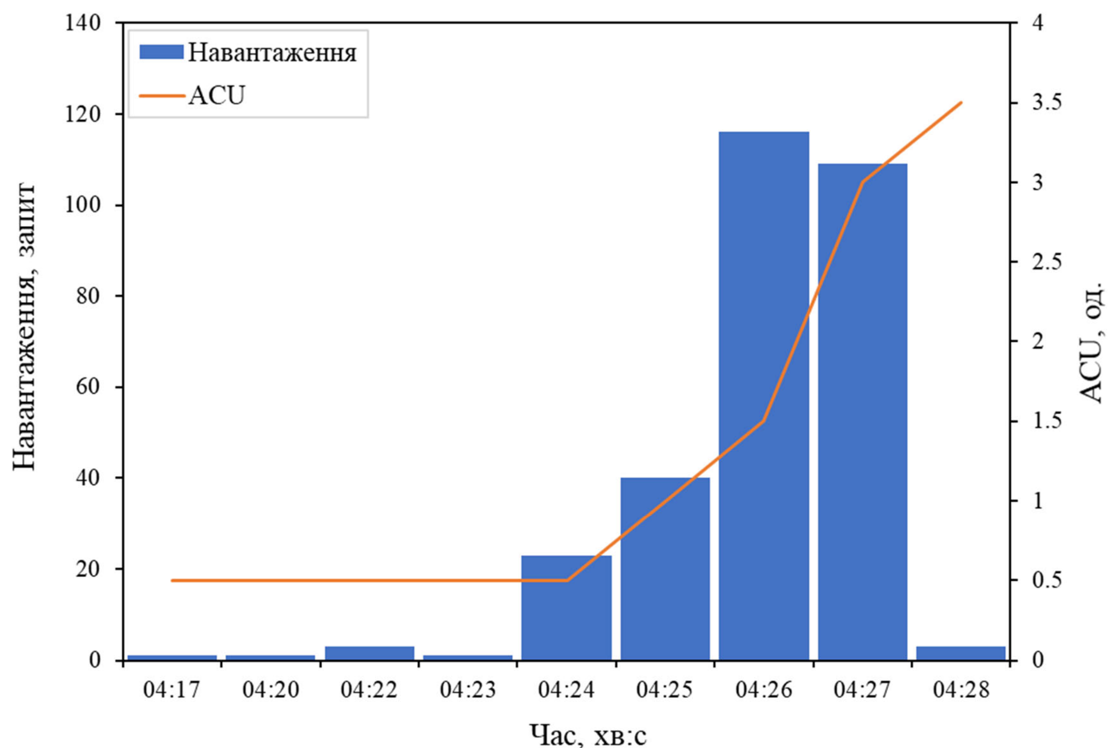

Рис. 3.5 – Автоматичне масштабування Aurora від запитів, холодна Lambda

Aurora Serverless v2 автоматично регулює свою потужність (в ACU) на основі обсягу та інтенсивності вхідних запитів. Зі збільшенням робочого навантаження Aurora повинна надавати додаткові ACU для підтримки стабільної пропускної здатності та мінімізації затримок запитів.

Під час тестування Lambda було зібрано дані щодо поточних потужностей Aurora (в ACU) та кількості запитів. Нажаль, гранульованість даних щодо ACU, наданих кластеру Aurora, лімітована AWS однією секундою.

Як видно з графіків запуску Lambda функцій з 256 МіБ ПДД, Aurora реактивно масштабує свої потужності у залежності від кількості запитів, проте спроб прогнозування навантаження зафіксовано не було.

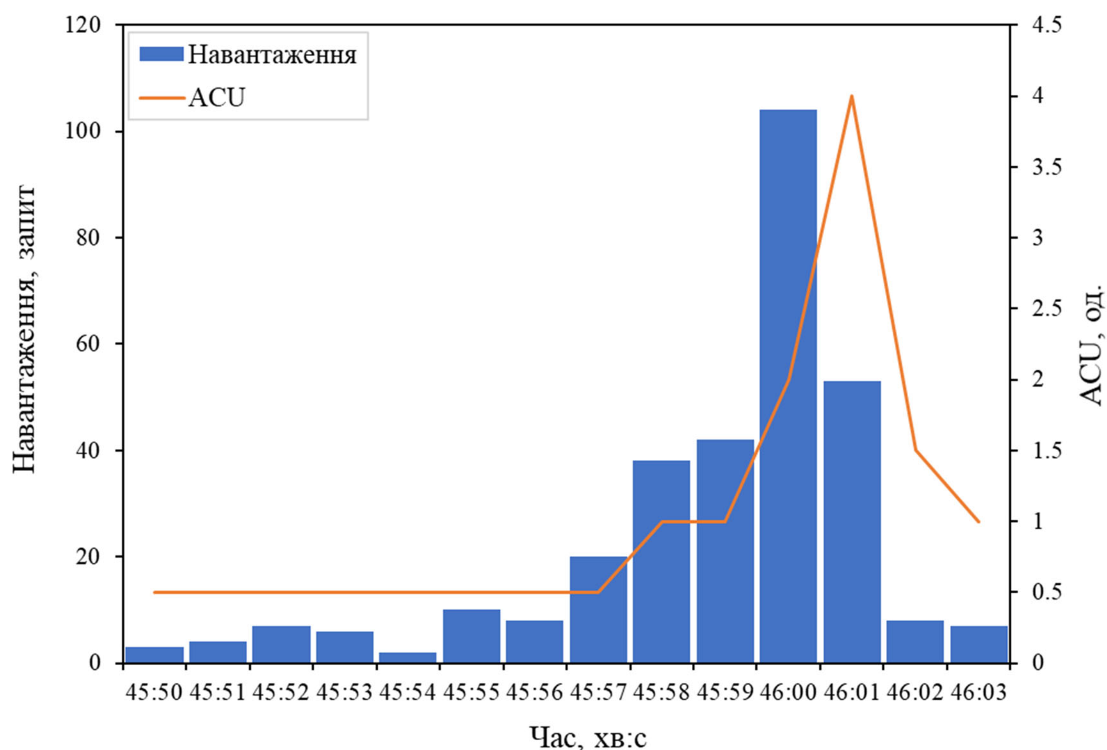

Рис. 3.6 – Автоматичне масштабування Aurora від запитів, тепла Lambda

Що стосується часу відповіді на запит, зв'язку масштабування Aurora з цим показником також помічено не було. Як можна побачити з графіку (рис.

3.7), функція декілька раз спостерігає підвищені часи відгуку від БД, що не викликає масштабування Aurora.

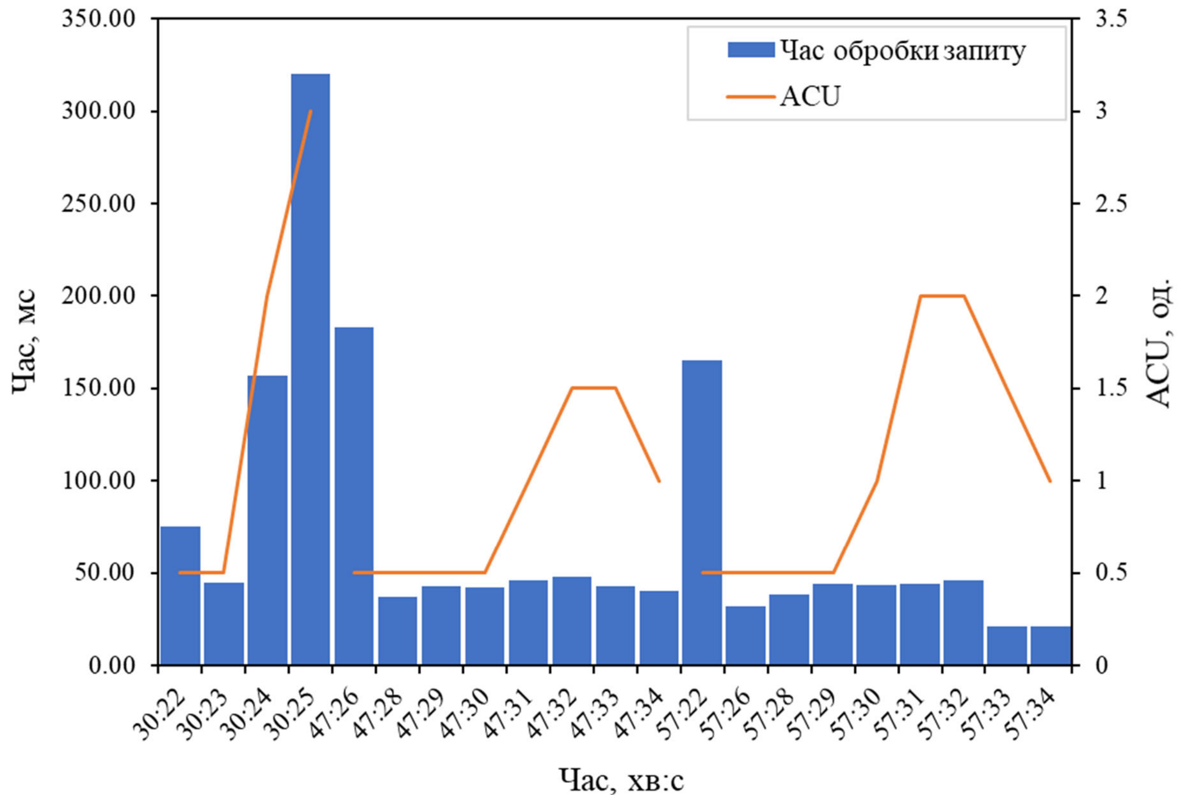

Рис. 3.7 – Автоматичне масштабування Aurora та час запиту

Додатково, аналізуючи обробку запитів, можна побачити, що мінливість цієї системи є дуже високою (c_v > 100%), тобто Aurora Serverless не забезпечує стабільного часу обробки запитів.

3.3 Висновки по розділу

Дані послідовно демонструють, що збільшення виділеної пам'яті AWS Lambda значно скорочує час як холодного, так і теплого запусків, тобто конфігурація пам’яті AWS Lambda є ключовим механізмом для підвищення швидкості виконання. Ефективність залежить від конкретних навантажень; вплив на ресурсоємні навантаження, такі як хешування за допомогою Argon2, є більшим, ніж на операції із залученням зовнішніх сервісів, як SSM. При цьому мінливість часу виконання функції залишається слабкою.

Для додатків, що вимагають низьких затримок, може знадобитися підтримка завжди теплих функцій, ретельне налаштування ресурсів і оптимізація їх архітектури (наприклад, кешування результатів в пам'яті) для мінімізації затримки використання зовнішніх сервісів.

Незважаючи на її обмеження, платформа AWS Lambda може використовуватися для ефективного масштабування web додатків у випадку, якщо час виконання функції та цінова політика задовольняють вимогам проекту.

В умовах значного паралелізму конфігурація пам’яті залишається ефективною, проте мінливість часу виконання функцій помітно збільшується.

Поведінка Aurora Serverless при масштабуванні є реактивною; хоча система здатна регулювати потужності у відповідь на навантаження, це відбувається не миттєво і може виникати погіршення продуктивності до введення додаткових потужностей до експлуатації.

Окрім цього мінливість часу обробки запитів при використанні Aurora Serverless є дуже високою, та система не реагує на підвищений час відгуку (окремо).

Цінова політика підтримання теплого кластеру Aurora Serverless у багато разів перевищує відокремлені ресурси Amazon RDS, а її мінливість робить її використання для масштабованих web додатків дуже сумнівною.

ВИСНОВКИ

Представлена робота спрямована на визначення практичних методів масштабування web додатків.

В роботі проведено аналіз сучасних методів та підходів до масштабування web додатків.

Виконано пошук даних та аналіз існуючих безсерверних рішень найкрупніших хмарних провайдерів, визначені їх особливості, переваги та недоліки.

Проведено експеримент на безсерверних платформах AWS, а саме AWS Lambda та Aurora Serverless для аналізу їх ефективності для розгортання web додатків. Критеріями слугували мінливість, виражена через коефіцієнт мінливості та простота розробки.

Визначено ефекти конфігурації AWS Lambda на час виконання безсерверних функцій у залежності від навантажень та використовуваних ресурсів екосистеми, таких як Amazon Parameter Store та Amazon Aurora.

Перевірено вплив паралельних запусків функцій Lambda на швидкість їх виконання.

Розглянуто автоматичне масштабування Aurora Serverless під час експерименту та залежність виділених ресурсів від кількості окремих параметрів.

Приведено рекомендації щодо використання платформи AWS для масштабованих web додатків.

Результати роботи можуть використовуватись у подальших дослідженнях хмарних платформ для порівняння з іншими рішеннями, підвищення ефективності та визначення оптимальних підходів масштабування окремих додатків.

За результатами роботи опубліковано наукову статтю у іноземному журналі [59].

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. World Internet Users and 2023 Population Stats. [Електронний ресурс] – Режим доступу: https://www.internetworldstats.com/stats.htm
2. Hashemi M., The Infrastructure Behind Twitter: Scale, Twitter Engineering. [Електронний ресурс] – Режим доступу: https://blog.twitter.com/engineering/en_us/topics/infrastructure/2017/the-infrastructure-behind-twitter-scale
3. Reducing Instagram's basic video compute time by 94 percent." [Електронний ресурс] – Режим доступу: https://engineering.fb.com/2022/11/04/video-engineering/instagram-video-processing-encoding-reduction/
4. Desai H., Unified Session for Analytical Events, Uber Blog. [Електронний ресурс] – Режим доступу: https://www.uber.com/en-CA/blog/unified-session-for-analytical-events
5. Boeve D. Throughput autoscaling: Dynamic sizing for Facebook.com / Daniel Boeve, Kiryong Ha, Anca Agape. [Електронний ресурс] – Режим доступу: https://engineering.fb.com/2020/09/14/networking-traffic/throughput-autoscaling/
6. Bondi A. B., Characteristics of scalability and their impact on performance, in Proceedings of the 2nd international workshop on Software and performance, Ottawa Ontario Canada: ACM, Sep. 2000, pp. 195–203. doi: 10.1145/350391.350432.
7. Gorton I., Foundations of scalable systems: designing distributed architectures, First edition. Beijing Sebastopol, CA: O'Reilly, 2022.
8. Carr D., Oregon Dumps Failed Health Insurance Exchange // InformationWeek. – 2014. – [Електронний ресурс] – Режим доступу: https://www.informationweek.com/it-sectors/oregon-dumps-failed-health-insurance-exchange.

9. Wayne A., Obamacare Website Costs Exceed $2 Billion, Study Finds // Bloomberg. – 2014. – [Електронний ресурс] – Режим доступу: https://www.bloomberg.com/news/articles/2014-09-24/obamacare-website-costs-exceed-2-billion-study-finds.

10. Agrawal D., El Abbadi A., Das S., Elmore A.J., Database Scalability, Elasticity, and Autonomy in the Cloud // Database Systems for Advanced Applications / J.X. Yu, M.H. Kim, R. Unland (Eds.) – Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. – Vol. 6587. – Pp. 2–15. – (Lecture Notes in Computer Science). doi: 10.1007/978-3-642-20149-3_2.

11. Modern applications at AWS – [Електронний ресурс] – Режим доступу: https://www.allthingsdistributed.com/2019/08/modern-applications-at-aws.html.

12. Google Cloud Pricing Calculator // [Електронний ресурс] – Режим доступу: https://cloud.google.com/products/calculator.

13. Amdahl G.M., Validity of the single processor approach to achieving large scale computing capabilities // Proceedings of the April 18-20, 1967, spring joint computer conference on - AFIPS ’67 (Spring). – Atlantic City, New Jersey: ACM Press, 1967. – P. 483. doi: 10.1145/1465482.1465560.

14. Kanev S., Profiling a warehouse-scale computer // Proceedings of the 42nd Annual International Symposium on Computer Architecture. – Portland Oregon: ACM, 2015. – Pp. 158–169. doi: 10.1145/2749469.2750392.

15. Mauro T., Adopting Microservices at Netflix: Lessons for Architectural Design // [Електронний ресурс] – Режим доступу: https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/.

16. Goldberg Y., Scaling Gilt: from Monolithic Ruby Application to Distributed Scala Micro-Services Architecture // [Електронний ресурс] – Режим доступу: https://www.infoq.com/presentations/scale-gilt/.

17. Villamizar M., Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud // 2015 10th Computing Colombian Conference (10CCC). – Bogota, Colombia: IEEE, 2015. – Pp. 583–590. doi: 10.1109/ColumbianCC.2015.7333476.
18. Nadareishvili I. Microservice architecture: aligning principles, practices, and culture. – First edition. – Sebastopol, CA: O'Reilly Media, Inc, 2016.
19. Mcrouter // [Електронний ресурс] – Режим доступу: https://github.com/facebook/mcrouter.
20. He X. Practical Lessons from Predicting Clicks on Ads at Facebook / X. He, H. Qian, J. Li, H. Chen // Proceedings of the Eighth International Workshop on Data Mining for Online Advertising. – New York NY USA: ACM, 2014. – Pp. 1–9. doi: 10.1145/2648584.2648589.
21. Borthakur D., et al. Apache hadoop goes realtime at Facebook // Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data. – Athens Greece: ACM, 2011. – Pp. 1071–1080. doi: 10.1145/1989323.1989438.
22. Aiyer A., Storage Infrastructure Behind Facebook Messages: Using HBase at Scale // IEEE International Conference on Data Engineering (ICDE). – Arlington, VA, USA, 2012.
23. Georgiou T., Li X., Migrating Messenger storage to optimize performance // [Online]. Available: https://engineering.fb.com/2018/06/26/core-infra/migrating-messenger-storage-to-optimize-performance/.
24. Matsunobu Y., MyRocks: A space- and write-optimized MySQL database // [Електронний ресурс] – Режим доступу: https://engineering.fb.com/2016/08/31/core-infra/myrocks-a-space-and-write-optimized-mysql-database/.
25. Zstandard - Fast real-time compression algorithm // [Електронний ресурс] – Режим доступу: https://github.com/facebook/zstd.

26. Borthakur D. Under the Hood: Building and open-sourcing RocksDB // [Електронний ресурс] – Режим доступу: https://engineering.fb.com/2013/11/21/core-infra/under-the-hood-building-and-open-sourcing-rocksdb/.
27. RocksDB: A Persistent Key-Value Store for Flash and RAM Storage // [Електронний ресурс] – Режим доступу: https://github.com/facebook/rocksdb.
28. Zhang W. Improving Write Performance of LSMT-Based Key-Value Store / W. Zhang, Y. Xu, Y. Li, D. Li // 2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS). – Wuhan, China: IEEE, 2016. – Pp. 553–560. doi: 10.1109/ICPADS.2016.0079.
29. Rabl T. Solving big data challenges for enterprise application performance management / T. Rabl, S. Gómez-Villamor, M. Sadoghi, V. Muntés-Mulero, H.-A. Jacobsen, S. Mankovskii // Proc. VLDB Endow. – 2012. – Vol. 5, No. 12. – Pp. 1724–1735. doi: 10.14778/2367502.2367512.
30. Kuhlenkamp J. Benchmarking scalability and elasticity of distributed database systems / J. Kuhlenkamp, M. Klems, O. Röss // Proc. VLDB Endow. – 2014. – Vol. 7, No. 12. – Pp. 1219–1230. doi: 10.14778/2732977.2732995.
31. Wang Z., Qian H., Li J., Chen H. Using restricted transactional memory to build a scalable in-memory database // Proceedings of the Ninth European Conference on Computer Systems. – Amsterdam, The Netherlands: ACM, 2014. – Pp. 1–15. doi: 10.1145/2592798.2592815.
32. Loesing S., Pilman M., Etter T., Kossmann D., On the Design and Scalability of Distributed Shared-Data Databases // Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. – Melbourne, Victoria, Australia: ACM, 2015. – Pp. 663–676. doi: 10.1145/2723372.2751519.
33. Salah Farrag A.A., Mahmoud S.A., El-Horbaty E.S.M. Intelligent cloud algorithms for load balancing problems: A survey // 2015 IEEE Seventh

International Conference on Intelligent Computing and Information Systems (ICICIS). – Cairo: IEEE, 2015. – Pp. 210–216. doi: 10.1109/IntelCIS.2015.7397223.

34. Mesbahi M., Rahmani A.M., Load Balancing in Cloud Computing: A State of the Art Survey // IJMECS. – 2016. – Vol. 8, No. 3. – Pp. 64–78. doi: 10.5815/ijmecs.2016.03.08.
35. Milani A.S., Navimipour N.J., Load balancing mechanisms and techniques in the cloud environments: Systematic literature review and future trends // Journal of Network and Computer Applications. – 2016. – Vol. 71. – Pp. 86–98. doi: 10.1016/j.jnca.2016.06.003.
36. Jafarnejad Ghomi E., Rahmani A.M., Nasih Qader N., Load-balancing algorithms in cloud computing: A survey // Journal of Network and Computer Applications. – 2017. – Vol. 88. – Pp. 50–71. doi: 10.1016/j.jnca.2017.04.007.
37. Ivanisenko I., Radivilova T., Survey of Major Load Balancing Algorithms in Distributed System. – 2019. doi: 10.48550/ARXIV.1904.05923.
38. Kanakala V.R., Reddy V.K., Karthik K., Performance analysis of load balancing techniques in cloud computing environment // 2015 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT). – Coimbatore, India: IEEE, 2015. – Pp. 1–6. doi: 10.1109/ICECCT.2015.7226052.
39. Keshvadi S., A Multi-Agent based Load Balancing System in IaaS Cloud Environment // IRATJ. – 2016. – Vol. 1, No. 1. doi: 10.15406/iratj.2016.01.00002.
40. Shen H., Sarker A., Yu L., Deng F., Probabilistic Network-Aware Task Placement for MapReduce Scheduling // 2016 IEEE International Conference on Cluster Computing (CLUSTER). – Taipei, Taiwan: IEEE, 2016. – Pp. 241–250. doi: 10.1109/CLUSTER.2016.48.
41. Shekhar S. Performance Interference-Aware Vertical Elasticity for Cloud-Hosted Latency-Sensitive Applications / S. Shekhar, H. Abdel-Aziz, A.

Bhattacharjee, A. Gokhale, X. Koutsoukos // 2018 IEEE 11th International Conference on Cloud Computing (CLOUD). – San Francisco, CA, USA: IEEE, 2018. – Pp. 82–89. doi: 10.1109/CLOUD.2018.00018.

42. Rossi F. Horizontal and Vertical Scaling of Container-Based Applications Using Reinforcement Learning / F. Rossi, M. Nardelli, V. Cardellini // 2019 IEEE 12th International Conference on Cloud Computing (CLOUD). – Milan, Italy: IEEE, 2019. – Pp. 329–338. doi: 10.1109/CLOUD.2019.00061.

43. Docker: Accelerated Container Application Development // [Електронний ресурс] – Режим доступу: https://www.docker.com/.

44. Jindal A., Podolskiy V., Gerndt M., Performance Modeling for Cloud Microservice Applications // Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering. – Mumbai, India: ACM, 2019. – Pp. 25–32. doi: 10.1145/3297663.3310309.

45. Chou D., Taiji: managing global user traffic for large-scale internet services at the edge // Proceedings of the 27th ACM Symposium on Operating Systems Principles. – Huntsville, Ontario, Canada: ACM, 2019. – Pp. 430–446. doi: 10.1145/3341301.3359655.

46. Social Hash: An Assignment Framework for Optimizing Distributed Systems Operations on Social Networks / A. Shalita, B. Karrer, I. Kabiljo та ін. // 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16). – Santa Clara, CA: USENIX Association, 2016. – [Електронний ресурс] – Режим доступу: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/shalita.

47. Yuan D., Joshi N., Jacobson D., Oberai P., Scryer: Netflix’s Predictive Auto Scaling Engine — Part 2 // Medium. – [Електронний ресурс] – Режим доступу: https://netflixtechblog.com/scryer-netflixs-predictive-auto-scaling-engine-part-2-bb9c4f9b9385.

48. Bak T., Kung F., Consistent caching mechanism in Titus Gateway // Medium. – [Online]. Available: https://netflixtechblog.com/consistent-caching-mechanism-in-titus-gateway-6cb89b9ce296.
49. Gibb C., Liu E., Wang Y. Introducing Kraken, an Open Source Peer-to-Peer Docker Registry // Uber Blog. – [Електронний ресурс] – Режим доступу: https://www.uber.com/en-NL/blog/introducing-kraken/.
50. Kraken. Go. Uber. – [Електронний ресурс] – Режим доступу: https://github.com/uber/kraken.
51. Cloud Market Growth Surge Continues in Q3 – Growth Rate Increases for the Fourth Consecutive Quarter // Synergy Research Group. – [Електронний ресурс] – Режим доступу: https://www.srgresearch.com/articles/cloud-market-growth-surge-continues-in-q3-growth-rate-increases-for-the-fourth-consecutive-quarter.
52. AWS Lambda Documentation // [Електронний ресурс] – Режим доступу: https://docs.aws.amazon.com/lambda/.
53. What is Amazon Aurora? - Amazon Aurora // [Електронний ресурс] – Режим доступу: https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/CHAP_AuroraOverview.html.
54. Azure Functions documentation // [Електронний ресурс] – Режим доступу: https://learn.microsoft.com/en-us/azure/azure-functions/.
55. MashaMSFT. Azure SQL documentation - Azure SQL // [Електронний ресурс] – Режим доступу: https://learn.microsoft.com/en-us/azure/azure-sql/?view=azuresql.
56. Cloud Run functions documentation // Google Cloud. – [Електронний ресурс] – Режим доступу: https://cloud.google.com/functions/docs.
57. Spanner documentation // Google Cloud. – https://cloud.google.com/spanner/docs.

58. argon2 // npm. – [Електронний ресурс] – Режим доступу: https://www.npmjs.com/package/argon2.

59. Kamieniev K., Kamienieva A., Lisitsyna I. Modern solutions and approaches to application scaling // Slovak international scientific journal. – 2024. – Vol. 88. – Pp. 7–13. doi: 10.5281/zenodo.13926460.

ДОДАТОК А

Код функції Lambda

```
exports.handler = async (event) => {

  let headers = {
    'Access-Control-Allow-Origin': '*',
    'Access-Control-Allow-Headers': 'Content-Type',
    'Access-Control-Allow-Methods': 'OPTIONS,POST'
  };

  let responseFn = (fnStatusCode, fnHeaders, fnBody) => {
    return {
      statusCode: fnStatusCode,
      headers: fnHeaders,
      body: fnBody
    };
  };

  let requestBody;

  if (event.body) {
    try {
      requestBody = JSON.parse(event.body);
    } catch (e) {
      // If the body is not in JSON format, return an error response
      return responseFn(400, headers, JSON.stringify({
        message: 'Invalid JSON format'
      }));
```

```
    }
  } else {
    // Return an error response if no body is found
    return responseFn(400, headers, JSON.stringify({
      message: 'No data received'
    }));
  }

  const emailRegex = /^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/; //[One or more allowed characters]@[One or more allowed characters].[At least two letters]
  const requestEmail = requestBody.email;
  if (!emailRegex.test(requestEmail))
    return responseFn(403, headers, JSON.stringify({
      message: 'Invalid email'
    }));

  const passwordRegex = /^.{1,}$/; //string has at least one character of any type
  const requestPassword = requestBody.password;
  if (!passwordRegex.test(requestPassword))
    return responseFn(403, headers, JSON.stringify({
      message: 'Invalid password'
    }));

  const {
    SSMClient,
    GetParametersCommand
  } = require("@aws-sdk/client-ssm");
```

```
const ssmClient = new SSMClient({
   region: "us-west-2"
});

const params = ['/db/user-auth-database/dbhost',
   '/db/user-auth-database/dbname',
   '/db/user-auth-database/username',
   '/db/user-auth-database/userpassword',
   '/db/user-auth-database/dbport',
   '/keys/private'
];

const command = new GetParametersCommand({
   Names: params,
   WithDecryption: true // Decrypt SecureString parameters
});
const parameters = {};

try {
   const data = await ssmClient.send(command);

   data.Parameters.forEach(param => {
      parameters[param.Name] = param.Value;
   });
} catch (error) {
   console.log(error);
   return responseFn(500, headers, JSON.stringify({
      error: 'Error fetching parameters'
   }));
```

```
}

const {
   Client
} = require('pg');
const client = new Client({
   host: parameters[params[0]], // Database endpoint
   user: parameters[params[2]], // Database username
   password: parameters[params[3]], // Database password
   database: parameters[params[1]], // Database name
   port: parameters[params[4]] //Database  port
});

let userID, hashedPassword;

try {
   const query = 'SELECT user_id, user_email, user_pswd_hash FROM
users.users WHERE user_email = $1';
   const dataArray = [requestEmail];

   await client.connect();
   const res = await client.query(query, dataArray);
   await client.end();

   if (res.rows.length < 1)
      return responseFn(401, headers, JSON.stringify('User doesn\'t exist'));

   userID = res.rows[0].user_id;
   hashedPassword = res.rows[0].user_pswd_hash;
```

```
} catch (error) {
   console.log(error);
   return responseFn(500, headers, JSON.stringify({
      error: 'Error reading from the database'
   }));
}

const argon2 = require('argon2');
try {
   const isMatch = await argon2.verify(hashedPassword,
requestPassword);

   if (!isMatch)
      return responseFn(401, headers, JSON.stringify('Wrong credentials'));
} catch (error) {
   console.log(error);
   return responseFn(500, headers, JSON.stringify({
      error: 'Error while checking the credentials'
   }));
}

const jwt = require('jsonwebtoken');
const privateKey = parameters[params[5]];
const serviceName = process.env.service; //service name stored in an
environmental variable

const payload = {
   userID: userID,
   service: serviceName,
```

```
    };
    const expiresIn = 3600; // Default expiration time is 1 hour

    try {
      const token = jwt.sign(payload, privateKey, {
        algorithm: 'RS256', //RS256 for RSA signing
        expiresIn: expiresIn
      });

      return responseFn(200, {
        ...headers,
        'Set-Cookie': `token=${token}; HttpOnly; Secure; Path=/; Max-Age=${expiresIn}; SameSite=Strict`
      }, JSON.stringify('JWT in cookie'));
    } catch (error) {
      console.log(error);
      return responseFn(500, headers, JSON.stringify({
        error: 'Error creating JWT'
      }));
    }

  };
```

ДОДАТОК Б

Графіки залежності кількості запитів від ACU

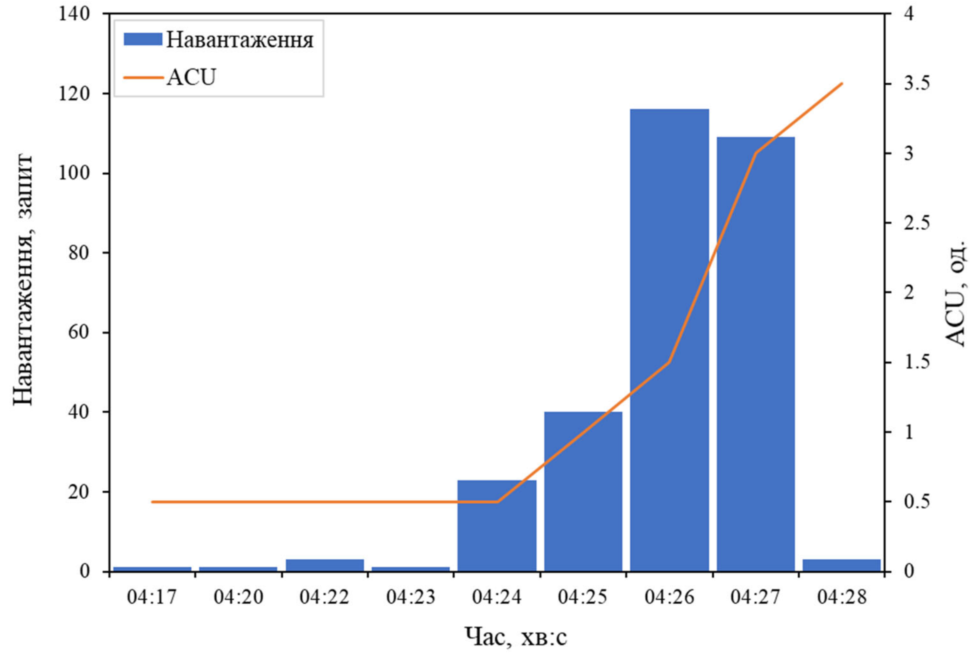

Рис. Б.1 – Холодна Lambda, 256 МіБ ПДД

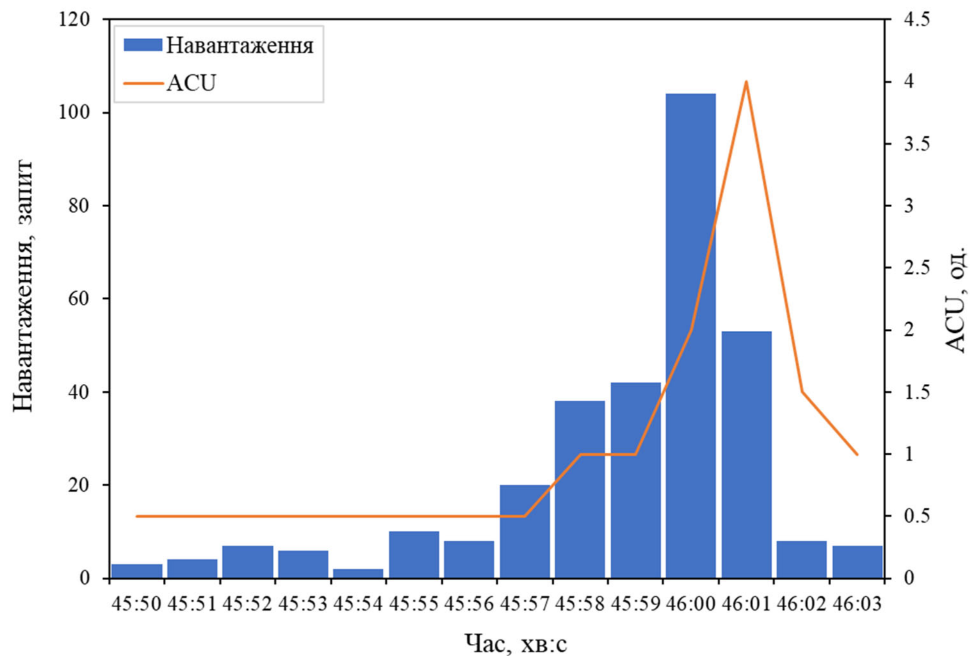

Рис. Б.2 – Тепла Lambda, 256 МіБ ПДД

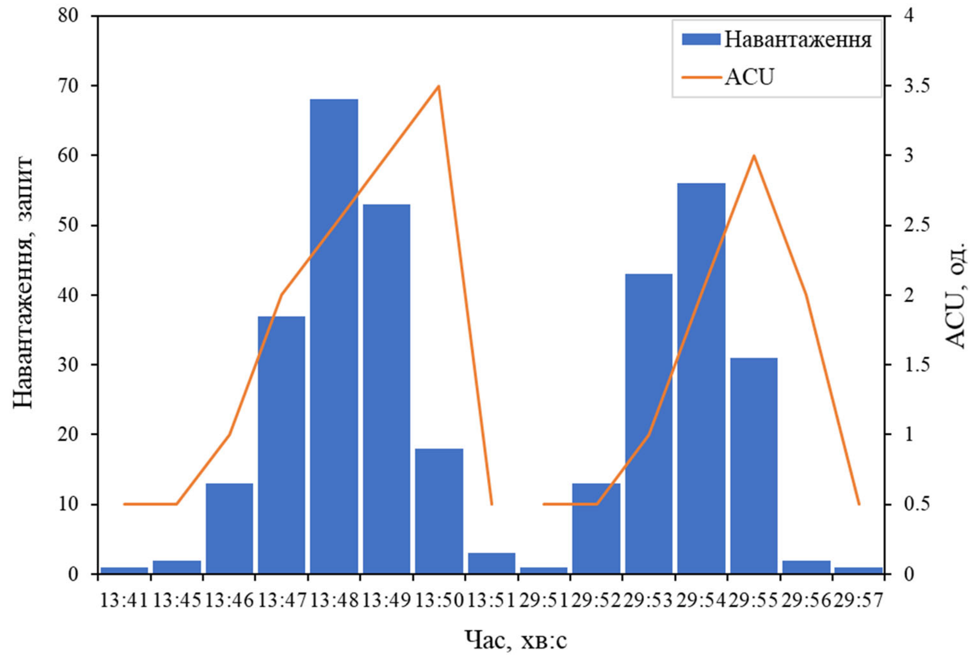

Рис. Б.3 – Холодна Lambda, 512 МіБ ПДД

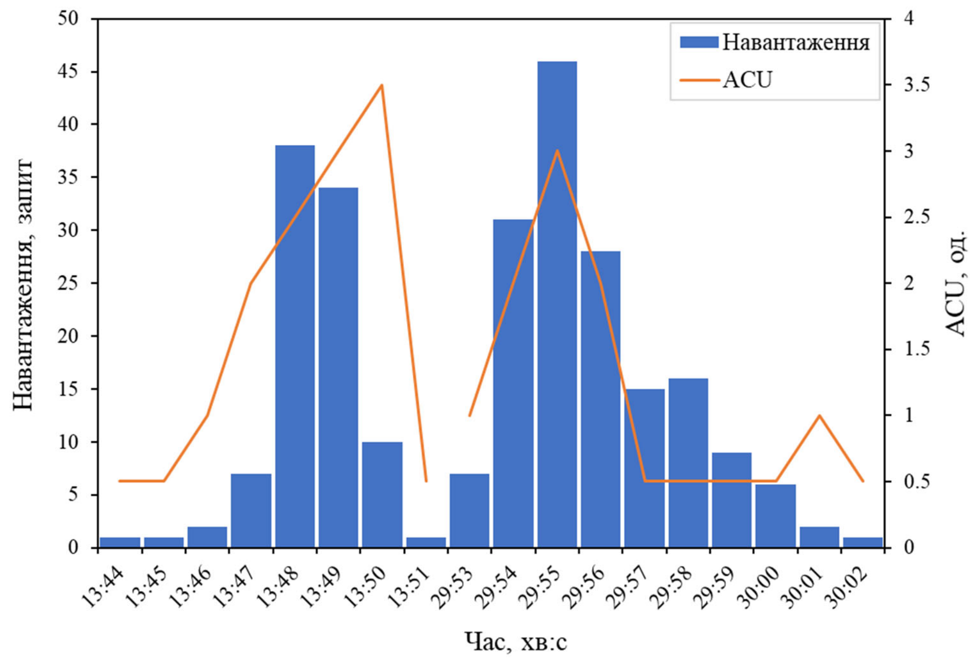

Рис. Б.4 – Тепла Lambda, 512 МіБ ПДД

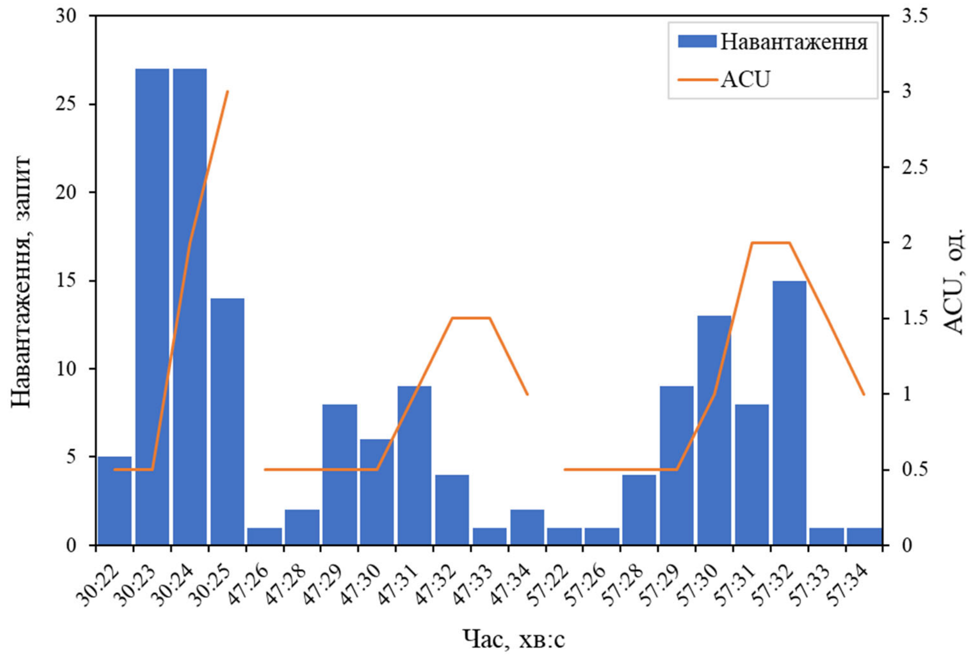

Рис. Б.5 – Холодна Lambda, 1024 МіБ ПДД

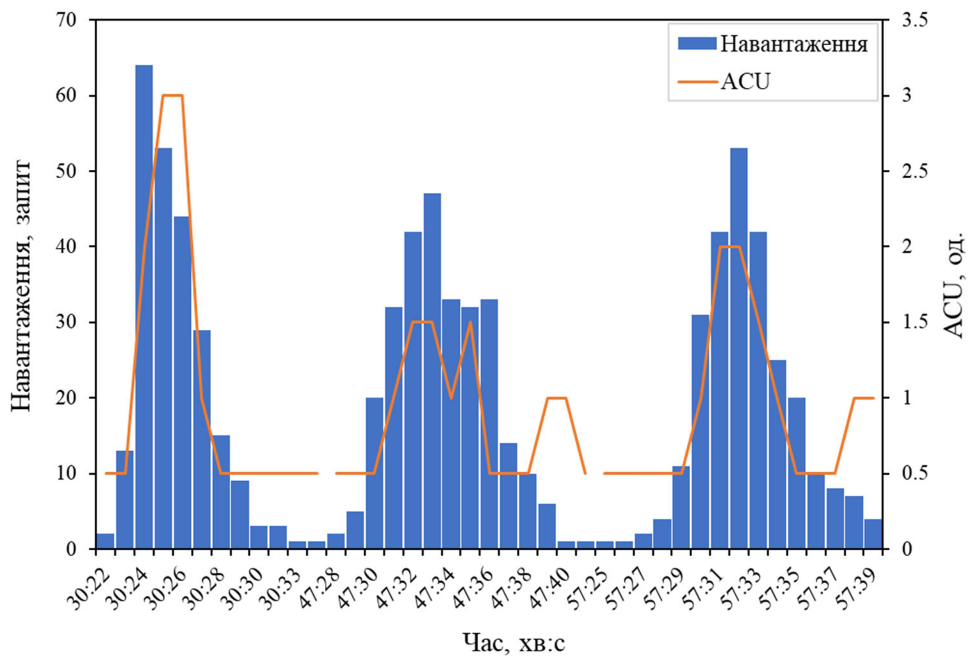

Рис. Б.6 – Тепла Lambda, 1024 МіБ ПДД

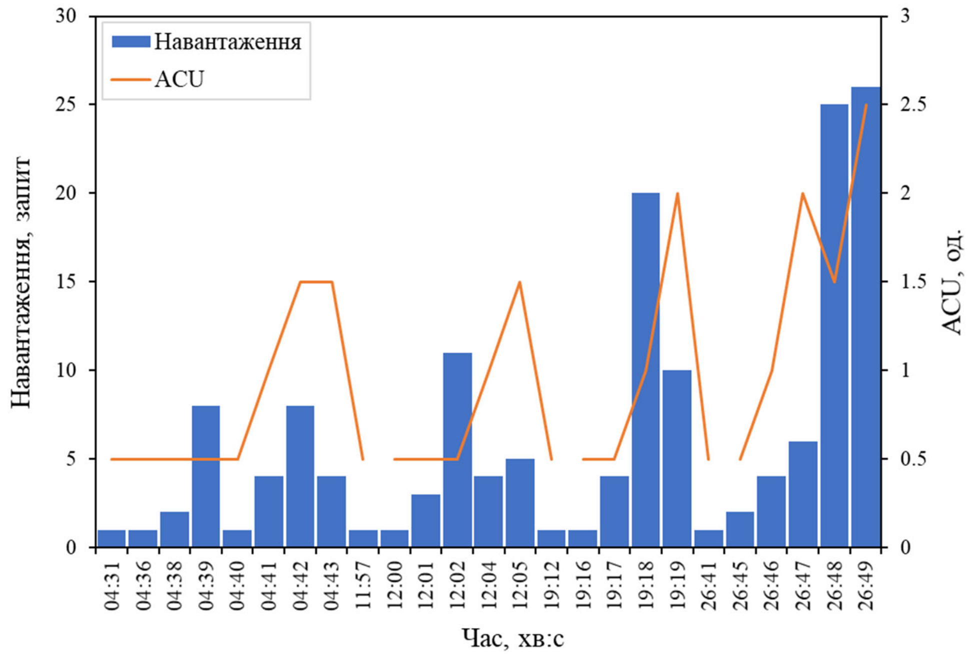

Рис. Б.7 – Холодна Lambda, 2048 МіБ ПДД

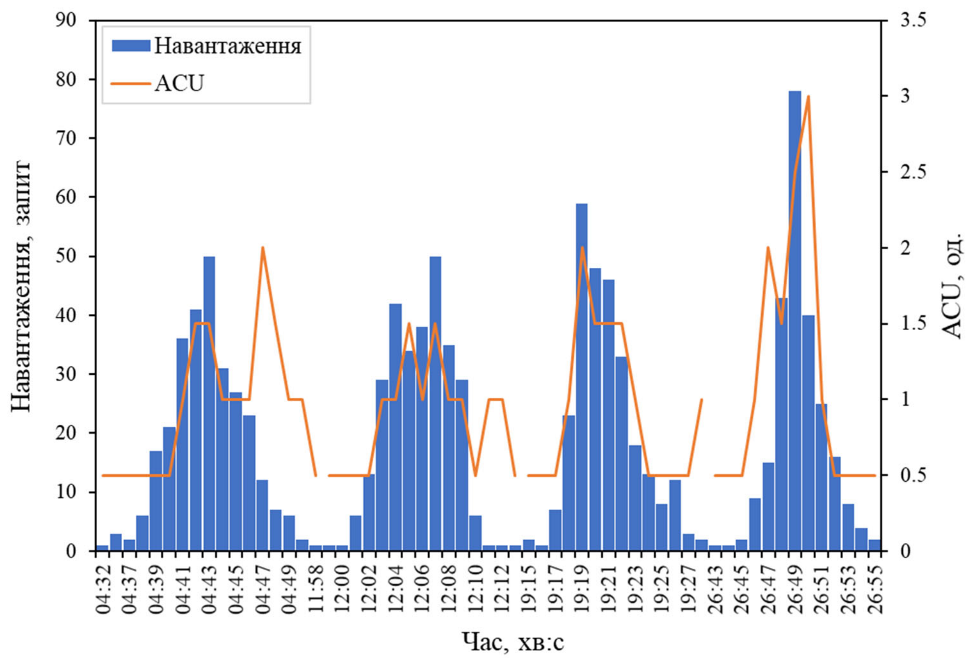

Рис. Б.8 – Тепла Lambda, 2048 МіБ ПДД

ДОДАТОК В

Графіки залежності часу обробки запитів від ACU

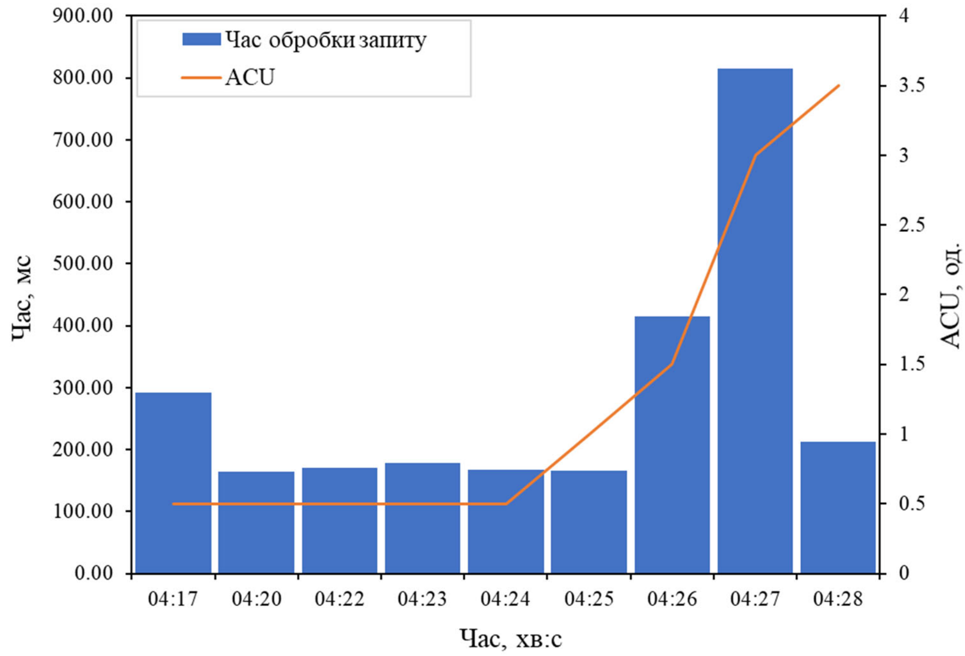

Рис. В.1 – Холодна Lambda, 256 МіБ ПДД

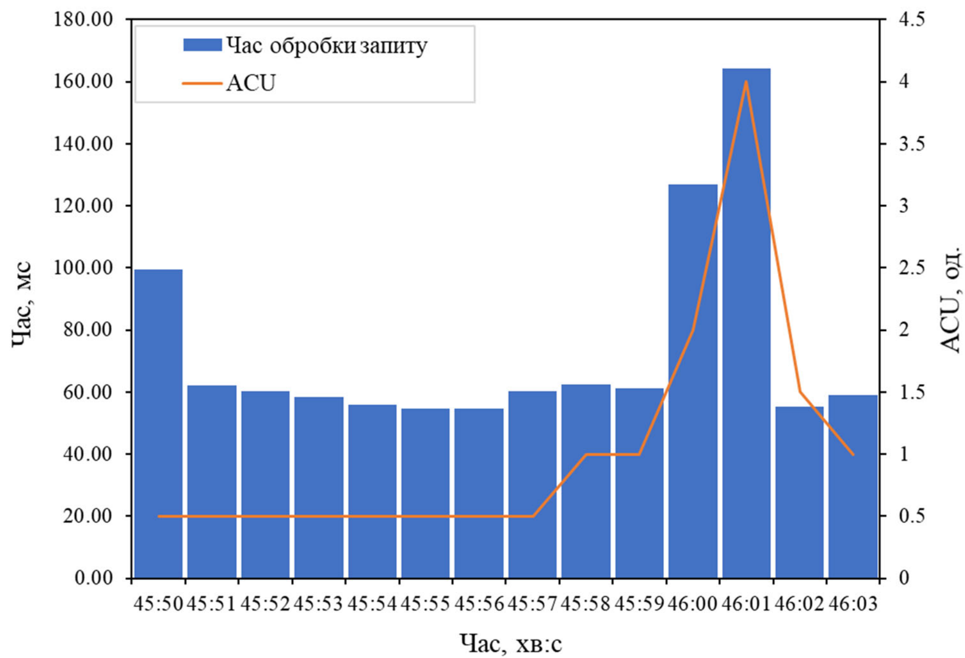

Рис. В.2 – Тепла Lambda, 256 МіБ ПДД

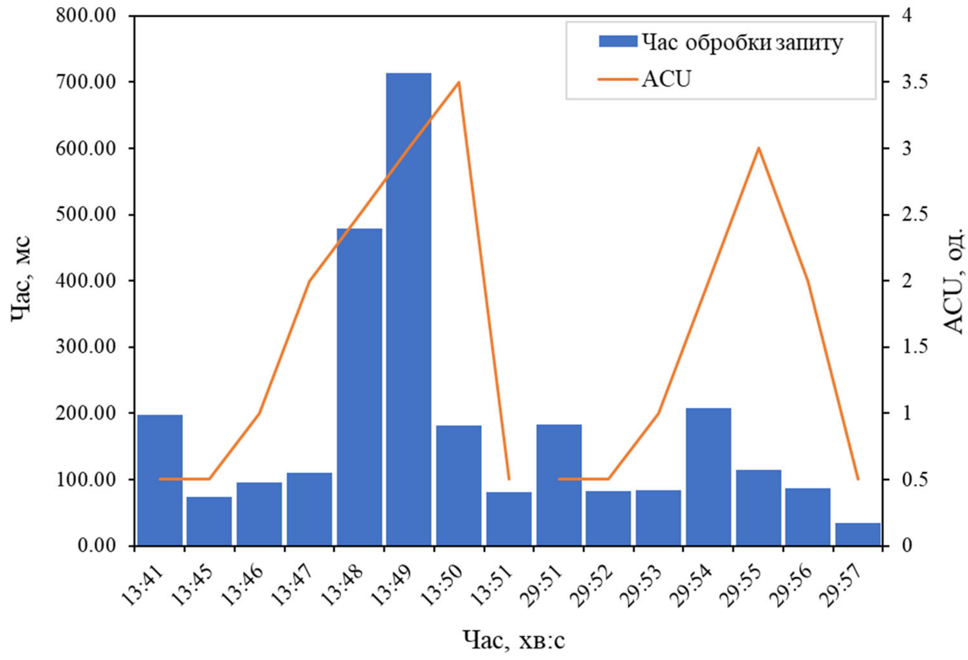

Рис. В.3 – Холодна Lambda, 512 МіБ ПДД

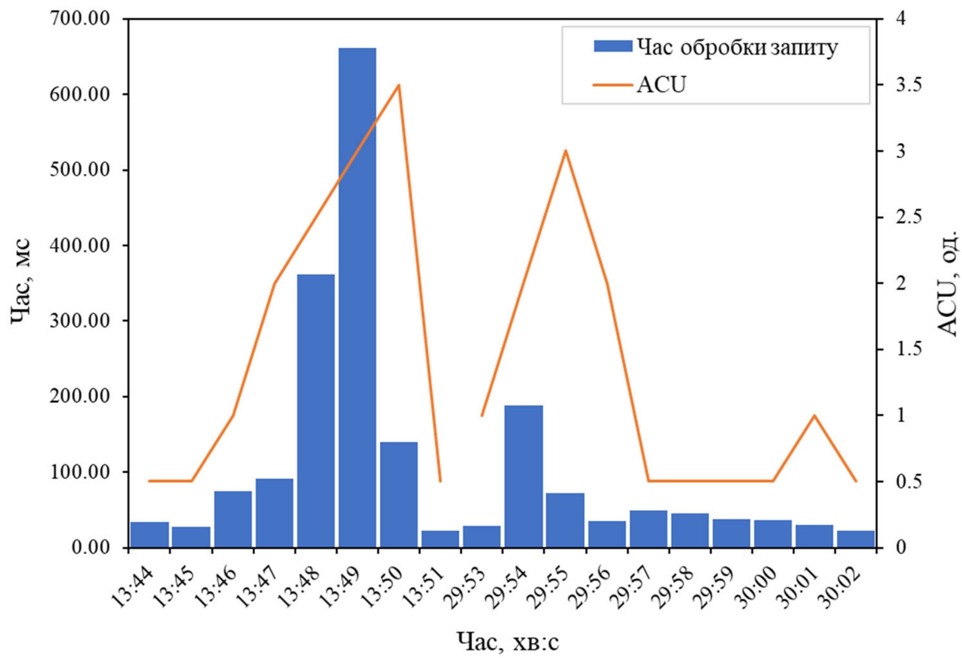

Рис. В.4 – Тепла Lambda, 512 МіБ ПДД

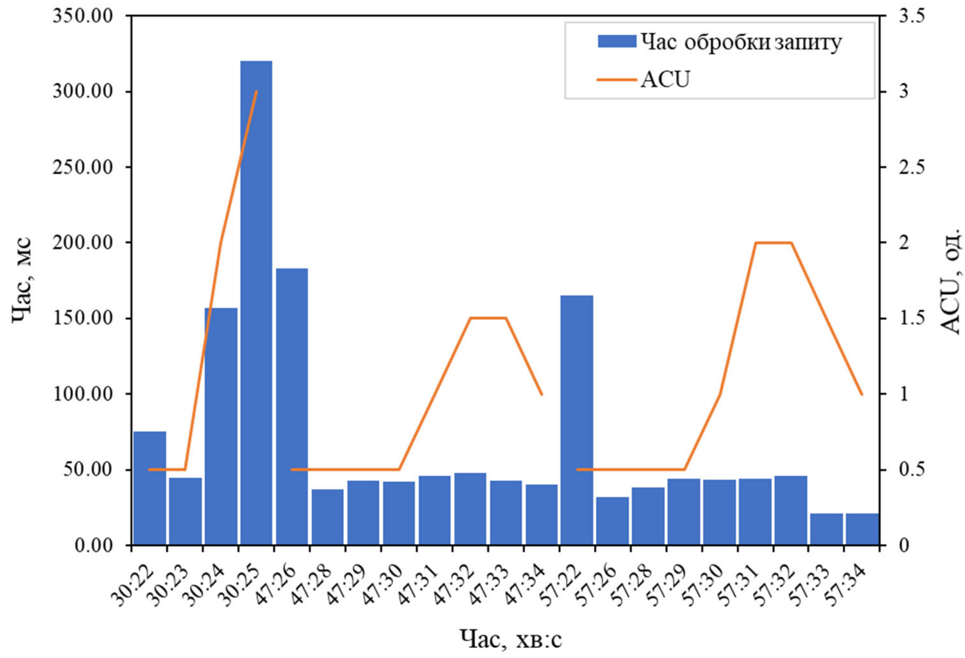

Рис. В.5 – Холодна Lambda, 1024 МіБ ПДД

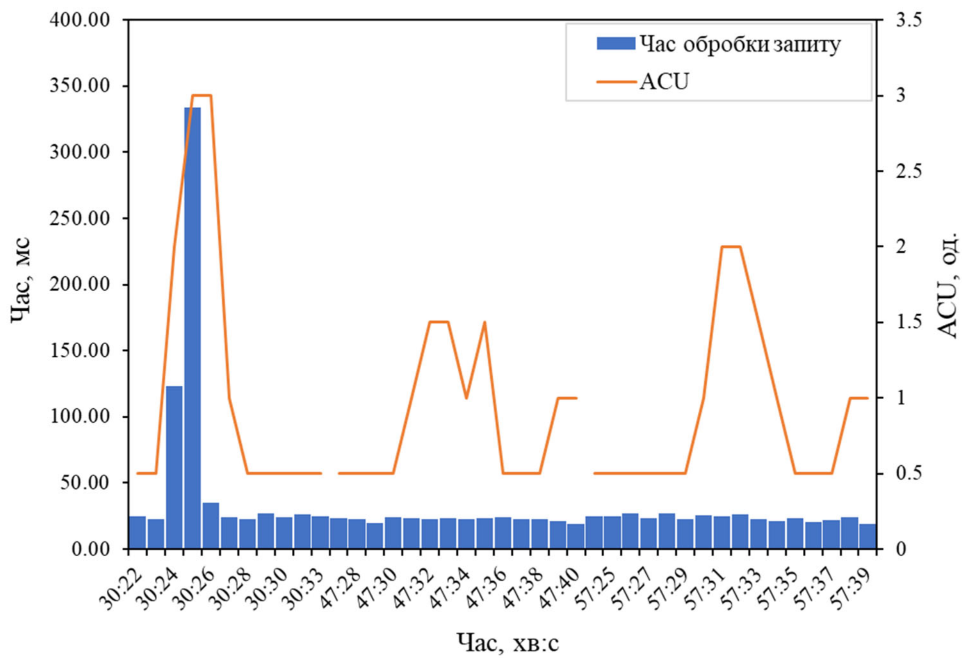

Рис. В.6 – Тепла Lambda, 1024 МіБ ПДД

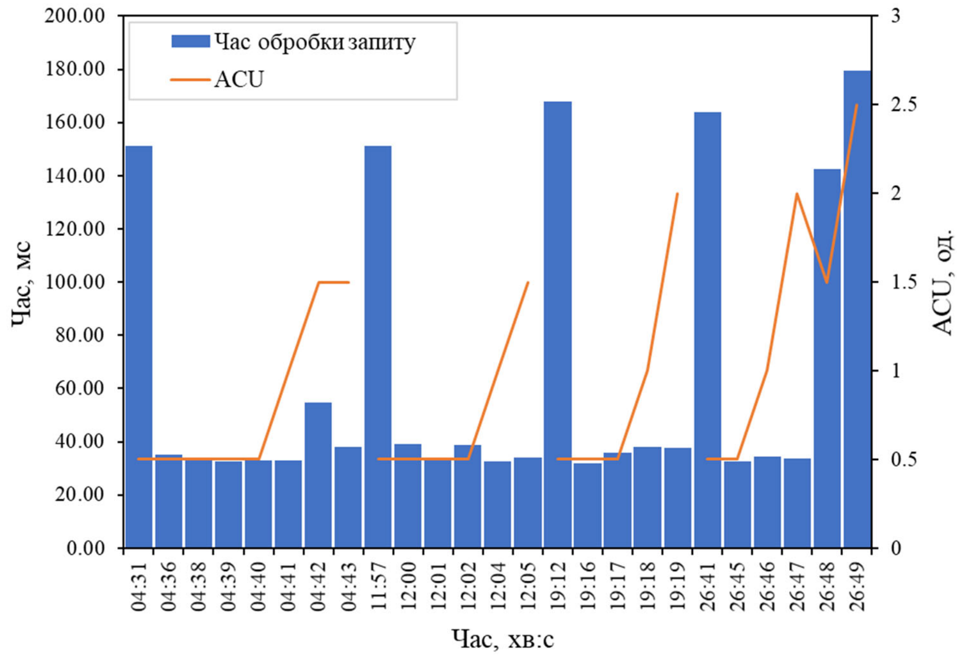

Рис. В.7 – Холодна Lambda, 2048 МіБ ПДД

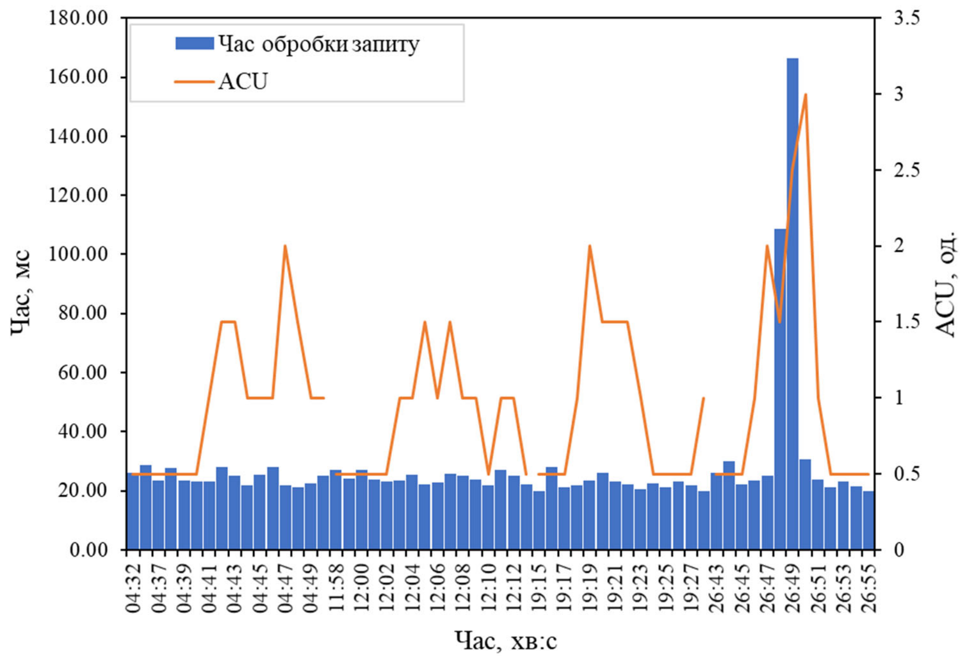

Рис. В.8 – Тепла Lambda, 2048 МіБ ПДД