

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ ЕКОНОМІЧНИЙ УНІВЕРСИТЕТ
ІМЕНІ ВАДИМА ГЕТЬМАНА

Навчально-науковий інститут
«Інститут інформаційних технологій в економіці»
Кафедра системного аналізу та кібербезпеки

ОСВІТНЬО-ПРОФЕСІЙНА ПРОГРАМА «СИСТЕМНИЙ АНАЛІЗ»

Галузь знань 12 «Інформаційні технології»

Спеціальність 124 «Системний аналіз»

Форма навчання: _____ очна (денна) _____
очна (денна), заочна, дистанційна

КВАЛІФІКАЦІЙНА БАКАЛАВРСЬКА РОБОТА

на тему «Моніторинг та моделювання надійності та масштабованості бекенд-системи»

здобувача _____ Корнєєва Владислава Олександровича _____
(прізвище, ім'я, по батькові) *(підпис)*

Науковий керівник: к.е.н _____ Агутін Михайло Михайлович _____
(науковий ступінь, учене звання, ПІБ) *(підпис)*

**Робота допущена до захисту перед екзаменаційною комісією
з атестації здобувачів вищої освіти (ЕК)**

Завідувач кафедри: д.ф.-м.н., проф. Джалладова І.А.

(підпис)

Київ 2026

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ ЕКОНОМІЧНИЙ УНІВЕРСИТЕТ
ІМЕНІ ВАДИМА ГЕТЬМАНА
Навчально-науковий інститут
«Інститут інформаційних технологій в економіці»
Кафедра системного аналізу та кібербезпеки

ОСВІТНЬО-ПРОФЕСІЙНА ПРОГРАМА «СИСТЕМНИЙ АНАЛІЗ»
СПЕЦІАЛЬНІСТЬ 124 «СИСТЕМНИЙ АНАЛІЗ»

ПОГОДЖЕНО

Керівник проєктної групи (гарант)
освітньо-професійної програми

(підпис)

_____ 2026 р.

ЗАТВЕРДЖУЮ

Завідувач кафедри системного
аналізу та кібербезпеки

_____ Джалладова І.А.
(підпис)

_____ 2026 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

здобувачу вищої освіти Корнєєву Владиславу Олександровичу

_____ очної (денної) _____ форми навчання

очної (денної), заочної, дистанційної

на підготовку кваліфікаційної бакалаврської роботи

на тему «Моніторинг та моделювання надійності та масштабованості бекенд-системи»

Тему затверджено наказом ректора Університету від «___» ___ 20__ р. No ___

Кваліфікаційна бакалаврська робота виконується на матеріалах підприємства «Galaxy4Games».

План кваліфікаційної бакалаврської роботи

Розділ 1	Теоретичні основи надійності та масштабованості бекенд-систем.
(назва розділу)	
Розділ 2	Аналіз та моделювання характеристик бекенд-системи веб-платформи galaxy4games.
(назва розділу)	
Розділ 3	Розробка засобів підвищення надійності та масштабованості бекенд-системи.
(назва розділу)	
Об'єкт дослідження:	Бекенд-система веб-платформи Galaxy4Games.
Предмет дослідження:	Методи підвищення продуктивності, надійності та масштабованості бекенд-систем за допомогою кешування, асинхронної обробки та оптимізації баз даних.
Мета кваліфікаційної бакалаврської роботи:	Підвищення продуктивності та масштабованості бекенд-системи шляхом виявлення архітектурних вузьких місць та оптимізації обробки запитів.

Конкретні завдання, які здобувач повинен виконати для досягнення поставленої мети:

У розділі 1	Дослідити особливості сучасних бекенд-архітектур, порівняти монолітний і мікросервісний підходи.
Систематизувати методи продуктивності й масштабованості: балансування, реплікацію, кешування, асинхронну обробку, пул з'єднань, індексування, патерни відмовостійкості.	
Розглянути метрики оцінювання продуктивності та обґрунтувати інструментарій навантажувального тестування.	
У розділі 2	Дослідити серверну архітектуру веб-платформи galaxy4games у стані As-Is.
Побудувати UML-моделі обробки запитів і сформулювати гіпотези щодо архітектурних обмежень.	
Провести навантажувальне тестування засобами Locust та отримати метрики продуктивності.	
Виявити «вузькі місця» та визначити граничну пропускну здатність системи.	
Виконати статистичну декомпозицію чинників зниження надійності та продуктивності.	
У розділі 3	Розробити систему моніторингу (Prometheus, Grafana, Alertmanager) за методологіями RED і USE.
Реалізувати механізми усунення вузьких місць системи.	
Обґрунтувати горизонтальне масштабування моделлю М/М/1 та провести повторне тестування.	
Оцінити ефективність рішень методом аналізу ієрархій (АНР).	

Завдання підготував
науковий керівник

_____ (підпис)

Агутін М.М.
(прізвище та ініціали)

« ____ » _____ 20__ р.

Завдання одержав
здобувач

_____ (підпис)

Корнєєв В.О.
(прізвище та ініціали)

« ____ » _____ 20__ р.

РЕФЕРАТ

Кваліфікаційна бакалаврська робота містить 69 сторінок основного тексту, 3 розділи, 17 рисунків, 19 таблиць, список використаних джерел із 30 найменувань, 12 додатки.

«Моніторинг та моделювання надійності та масштабованості бекенд-системи»

Об'єктом дослідження є бекенд-система веб-платформи Galaxy4Games.

Предметом дослідження є методи підвищення продуктивності, надійності та масштабованості бекенд-систем за допомогою кешування, асинхронної обробки та оптимізації баз даних.

Мета кваліфікаційної бакалаврської роботи - підвищення продуктивності та масштабованості бекенд-системи шляхом виявлення архітектурних вузьких місць та оптимізації обробки запитів на основі системно-аналітичного дослідження її архітектури та характеристик продуктивності під навантаженням.

Відповідно до поставленої мети були визначені такі завдання:

Дослідити особливості сучасних бекенд-архітектур, порівняти монолітний і мікросервісний підходи та систематизувати методи забезпечення продуктивності й масштабованості;

Обґрунтувати метрики оцінювання продуктивності та інструментарій навантажувального тестування;

Дослідити серверну архітектуру веб-платформи Galaxy4Games у стані As-Is та побудувати UML-моделі процесів обробки запитів;

Провести навантажувальне тестування засобами Locust, виявити «вузькі місця» й визначити граничну пропускну здатність системи;

Виконати статистичну декомпозицію чинників зниження надійності та продуктивності;

Розробити систему моніторингу (Prometheus, Grafana, Alertmanager) за методологіями RED і USE та реалізувати механізми усунення вузьких місць;

Обґрунтувати горизонтальне масштабування моделлю масового обслуговування М/М/1 і оцінити ефективність запропонованих рішень методом аналізу ієрархій (АНР).

Методи дослідження: UML-моделювання для формалізації процесів обробки запитів; навантажувальне тестування засобами Locust для отримання емпіричних метрик продуктивності; статистичний аналіз для декомпозиції чинників деградації; модель масового обслуговування M/M/1 для кількісного обґрунтування горизонтального масштабування; метод аналізу ієрархій для порівняльного оцінювання альтернативних архітектурних рішень.

Практичне значення отриманих результатів. Запропонований комплекс заходів - кеш Redis за схемою Cache-Aside, винесення повільних операцій у чергу Apache Kafka, оптимізація ORM-запитів і пулу з'єднань - забезпечив зростання пропускної здатності системи з 332 до 1 278,8 RPS та скорочення медіанного часу відповіді до 860 мс без помилок. Обґрунтована схема горизонтального масштабування дає близько дванадцятикратного приросту щодо стартового рівня. Розроблений аналітичний цикл є відтворюваним і придатним для дослідження інших мікросервісних бекенд-систем зі схожим профілем навантаження.

Рік виконання кваліфікаційної бакалаврської роботи - 2026.

Рік захисту роботи - 2026.

Ключові слова: бекенд-система, надійність, масштабованість, навантажувальне тестування, кешування, моніторинг, метод аналізу ієрархій.

ВІДГУК

про кваліфікаційну бакалаврську роботу здобувача
навчально-наукового інституту «Інститут інформаційних технологій в економіці»
освітньо-професійної програми «Системний аналіз»
Корнєєва Владислава Олександровича
на тему «Моніторинг та моделювання надійності та масштабованості бекенд-системи»

1. *Актуальність теми.* Тема дослідження здобувача Корнєєва Владислава обґрунтована необхідністю підвищення продуктивності бекенд-систем, що безпосередньо впливає на користувацький досвід та життєздатність публічних веб-ресурсів. Автор обґрунтовує необхідність системного підходу до архітектури, оскільки раптова втрата продуктивності під навантаженням зумовлена не помилками логіки ІТ-систем, а розривом між задекларованою та фактичною конфігурацією механізмів кешування та балансування.

2. *Позитивні риси кваліфікаційної роботи.* В кваліфікаційній роботі Корнєєва Владислава було проведено всебічний аналіз бекенд-системи Веб-платформи компанії Galaxy4Games. При цьому автор застосував сучасні методи дослідження: *UML-моделювання* для формалізації процесів обробки запитів; *емпіричне навантажувальне тестування* засобами Locust для отримання метрик продуктивності; *статистичне дослідження* для виявлення чинників втрати продуктивності бекенд-системи; застосування *математичного апарату теорії масового обслуговування* (модель М/М/1) для кількісного прогнозування поведінки системи; *метод аналізу ієрархій* для порівняльного оцінювання альтернативних архітектурних рішень.

3. *Наявність самостійних розробок автора.* Здобувач Корнєєв Владислав самостійно розробив та обґрунтував: необхідність використання *трирівневої системи моніторингу* на базі додатків Prometheus, Grafana та Alertmanager; програмну реалізацію механізмів оптимізації кешування та асинхронної обробки черг, маршрутизації запитів основною та резервною базами даних; логіку автоматичного *масштабування в хмарному середовищі* AWS Auto Scaling Group, яка базується на асиметричних параметрах для забезпечення високої доступності бекенд-системи.

4. *Цінність теоретичних висновків та практичних рекомендацій.* Робота з теоретичної та практичної точки зору пропонує інтегрований аналітичний підхід, придатний для дослідження будь-яких мікросервісних систем, що підтверджується результатами аналітичного дослідження: збільшення пропускну здатності бекенд-системи в 4 рази, скорочення агрегованого медіанного часу відгуку системи до 860мс; зростання продуктивності порівняно з початковим станом бекенд-системи майже в 12 разів.

5. *Наявність недоліків.* До недоліків можна віднести недостатньо детальний розгляд питань безпеки при передачі даних між репліками баз даних. Також у тексті присутні поодинокі стилістичні похибки та специфічні англомовні терміни, що потребують детального пояснення.

6. *Загальна оцінка кваліфікаційної бакалаврської роботи та її допущення до захисту перед ЕК.* В цілому робота Корнєєва Владислава відповідає вимогам до кваліфікаційних бакалаврських робіт за спеціальністю «Системний аналіз», виконана на високому теоретико-аналітичному рівні, містить завершені прикладні рішення та може бути допущена до захисту перед екзаменаційною комісією.

Науковий керівник: доцент кафедри системного аналізу
та кібербезпеки, кандидат економічних наук

“ _____ ” _____ 2026 р.

_____ М.М.Агутін

Рецензія

на кваліфікаційну бакалаврську роботу здобувача вищої освіти

Корнєєва Владислава Олександровича

(прізвище, ім'я, по батькові)

Тема: Моніторинг та моделювання надійності та масштабованості бекенд-системи

Актуальність теми кваліфікаційної роботи і доцільність її розроблення

Тема кваліфікаційної роботи є актуальною та відповідає сучасним потребам галузі інформаційних технологій і змісту спеціальності 124 «Системний аналіз». Бекенд-системи публічних веб-ресурсів функціонують в умовах непередбачуваного навантаження, за якого падіння продуктивності настає раптово - у момент, коли кількість одночасних запитів перетинає певну порогову межу. Як обґрунтовано в роботі, першопричиною такого падіння найчастіше є не помилки бізнес-логіки, а розрив між задекларованою архітектурою та її фактичною конфігурацією, коли механізми кешування, асинхронної обробки чи балансування присутні в системі, але задіяні лише частково.

Доцільність розроблення теми зумовлена тим, що архітектурні рішення у таких системах нерідко приймаються без формальної верифікації їх поведінки під навантаженням. Запропоноване автором поєднання UML-моделювання реального стану системи, навантажувального тестування та багатокритеріального оцінювання рішень у єдиному відтворюваному аналітичному циклі усуває цю прогалину і має як наукову, так і прикладну цінність. Робота виконана на матеріалах реального підприємства (веб-платформа Galaxy4Games), що додатково підтверджує практичну значущість обраного напрямку.

Якість проведеного дослідження

Дослідження виконано на належному науковому рівні, відзначається логічною завершеністю та послідовністю викладу. Робота має чітку трирівневу структуру: від систематизації теоретичних засад надійності й масштабованості бекенд-систем - через аналіз і моделювання характеристик реальної системи у стані As-Is - до розроблення та кількісного обґрунтування засобів її вдосконалення.

Автор коректно застосував комплекс системно-аналітичних методів: UML-моделювання для формалізації процесів обробки запитів, навантажувальне тестування засобами Locust для отримання емпіричних метрик, статистичну декомпозицію чинників деградації, модель масового обслуговування М/М/1 для обґрунтування горизонтального масштабування та метод аналізу ієрархій (АНР) для порівняльного оцінювання альтернатив. Особливо слід відзначити методологічну строгість: три гіпотези щодо архітектурних обмежень спочатку формулюються на рівні UML-моделей, а потім кількісно перевіряються експериментом, що забезпечує обґрунтованість висновків. Узгодженість суджень у методі АНР підтверджена розрахунком ($CR = 0,019$ за допустимого порога $0,10$), що свідчить про коректність застосування математичного апарату.

Позитивні риси кваліфікаційної бакалаврської роботи

До основних позитивних рис роботи слід віднести:

Практичну спрямованість і завершеність: запропонований комплекс заходів (кеш Redis за схемою Cache-Aside, винесення повільних операцій у чергу Apache Kafka, оптимізація ORM-запитів і пулу з'єднань) доведено до конкретних конфігурацій та фрагментів коду, придатних до прямого впровадження;

Вимірюваний результат: пропускну здатність системи підвищено з 332 до 1 278,8 RPS, а медіанний час відповіді скорочено до 860 мс за відсутності помилок, що підтверджено повторним навантажувальним тестуванням за тих самих умов;

Поєднання експериментального підходу з математичним моделюванням: перспективи масштабування обґрунтовано моделлю M/M/1, що дає близько дванадцятикратного приросту щодо стартового рівня;

Комплексність рішення з моніторингу (Prometheus, Grafana, Alertmanager) за методологіями RED і USE з диференціацією рівнів сповіщень;

Відтворюваність розробленого аналітичного циклу, придатного для дослідження інших мікросервісних бекенд-систем зі схожим профілем навантаження.

Зауваження

Принципових зауважень, що знижували б цінність роботи та обґрунтованість отриманих результатів, немає. Як побажання можна відзначити:

Окремі теоретичні положення першого розділу доцільно було б супроводити ширшим переліком порівняльних прикладів із практики;

Опис очікуваних організаційних та вартісних наслідків впровадження горизонтального масштабування на AWS-інфраструктурі міг би бути більш розгорнутим;

У тексті трапляються поодинокі технічні й стилістичні неточності в оформленні, що мають редакційний характер і не впливають на зміст роботи.

Зазначені зауваження мають характер рекомендацій і не знижують загальної позитивної оцінки роботи.

Практична значимість висновків і рекомендацій

Запропонований комплекс архітектурних рішень забезпечив майже чотирикратне зростання пропускну здатності та скорочення часу відповіді на порядок без появи помилок під навантаженням. Результати доведено до рівня готових конфігурацій і програмних фрагментів, що уможливорює їх безпосереднє впровадження у досліджувану систему. Розроблений аналітичний цикл - від виявлення вузьких місць засобами тестування й моніторингу до їх усунення та підтвердження масштабування розрахунком - не прив'язаний до конкретного застосунку і може бути використаний для інших мікросервісних систем із переважно читаючим профілем навантаження.

Кваліфікаційна бакалаврська робота Корнеєва Владислава Олександровича відповідає вимогам, що висуваються до робіт цього рівня, а її автор заслуговує на присвоєння ступеня бакалавра за спеціальністю 124 «Системний аналіз». Робота може бути оцінена на «відмінно».

Рецензент

Директор ТОВ «Galaxy4Games»

(підпис)

Нестеренко О. С.

(прізвище, ініціали)

“18” червня 2026 р.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ	3
ВСТУП.....	5
РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ НАДІЙНОСТІ ТА МАСШТАБОВАНOSTІ БЕКЕНД-СИСТЕМ	7
1.1. Особливості побудови сучасних веб-систем та бекенд-архітектур	8
1.2. Методи забезпечення продуктивності та масштабованості бекенд-систем.	13
1.3. Методи оцінювання продуктивності інформаційних систем	19
РОЗДІЛ 2. АНАЛІЗ ТА МОДЕЛЮВАННЯ ХАРАКТЕРИСТИК БЕКЕНД-СИСТЕМИ ВЕБ-ПЛАТФОРМИ GALAXY4GAMES	23
2.1. Дослідження об'єкта та UML-моделювання процесів обробки запитів	24
2.2. Навантажувальне тестування та виявлення «вузьких місць» у системі.....	30
2.3. Статистичний аналіз та декомпозиція чинників зменшення надійності	34
РОЗДІЛ 3. РЕАЛІЗАЦІЯ ЗАСОБІВ ПІДВИЩЕННЯ НАДІЙНОСТІ ТА МАСШТАБОВАНOSTІ БЕКЕНД-СИСТЕМИ.....	38
3.1. Моніторинг бекенд-системи.....	39
3.2 Підвищення надійності та масштабованості бекенд-системи.....	45
3.3 Оцінювання ефективності запропонованих рішень	59
ВИСНОВКИ.....	64
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	67
ДОДАТКИ	70

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

ACID (Atomicity, Consistency, Isolation, Durability) - набір властивостей транзакцій баз даних.

AHP (Analytic Hierarchy Process) - метод аналізу ієрархій.

ALB (Application Load Balancer) - балансувальник навантаження AWS.

API (Application Programming Interface) - програмний інтерфейс застосунку.

AWS (Amazon Web Services) - хмарна платформа Amazon.

AWS RDS (Relational Database Service) - керований сервіс реляційних баз даних AWS.

Cache-Aside - шаблон кешування з перевіркою кешу перед зверненням до бази даних.

CHR (Cache Hit Ratio) - коефіцієнт влучань у кеш.

CR (Consistency Ratio) - коефіцієнт узгодженості в методі АНР.

CV (Coefficient of Variation) - коефіцієнт варіації.

Docker - платформа контейнеризації застосунків.

Fail Fast - стратегія швидкої відмови системи.

GET - HTTP-метод отримання даних.

GIL (Global Interpreter Lock) - глобальне блокування інтерпретатора Python.

Grafana - система візуалізації метрик і моніторингу.

JSON (JavaScript Object Notation) - формат обміну даними.

MTTD (Mean Time To Detect) - середній час виявлення проблеми.

MVP (Minimum Viable Product) - мінімально життєздатний продукт.

ORM (Object-Relational Mapping) - технологія об'єктно-реляційного відображення.

POST - HTTP-метод створення або надсилання даних.

Primary - основний сервер бази даних.

Prometheus - система збору та зберігання метрик.

Query Duration - тривалість виконання SQL-запиту.

Read Replica - репліка бази даних для операцій читання.

REST (Representational State Transfer) - архітектурний стиль вебсервісів.

RPS (Requests Per Second) - кількість запитів за секунду.

Semaphore - механізм обмеження конкурентного доступу до ресурсів.

SMTP (Simple Mail Transfer Protocol) - протокол передавання електронної пошти.

SQLAlchemy - ORM-бібліотека для Python.

SSL/TLS (Secure Sockets Layer / Transport Layer Security) - криптографічні протоколи захисту даних.

TCP (Transmission Control Protocol) - транспортний протокол передачі даних.

Throughput - пропускна здатність системи.

TTL (Time To Live) - час життя запису в кеші.

UML (Unified Modeling Language) - уніфікована мова моделювання.

WAL (Write-Ahead Logging) - механізм журналювання транзакцій PostgreSQL.

ВСТУП

Бекенд-система, що обслуговує публічний веб-ресурс, функціонує в умовах, де навантаження є непередбачуваним, а деградація продуктивності проявляється не поступово, а раптово - у момент, коли кількість одночасних запитів перетинає певну порогову межу. Практика показує, що першопричиною такої деградації рідко є помилки у бізнес-логіці: частіше вона зумовлена розривом між задекларованою архітектурою та її фактичною конфігурацією. Механізми кешування, асинхронної обробки та балансування навантаження можуть бути присутні в системі, проте задіяні частково або не для тих операцій, де їх вплив був би найвідчутнішим. Виявлення такого розриву потребує не точкового налаштування окремих компонентів, а системно-аналітичного підходу - формальної декомпозиції архітектури, кількісного вимірювання характеристик під навантаженням і обґрунтованого вибору між конкуруючими рішеннями.

Описана проблема є характерною не лише для великих платформ з мільйонною аудиторією, а й для систем з відносно вузьким призначенням - корпоративних веб-ресурсів, каталогів, форм зворотного зв'язку, - які в умовах пікового трафіку демонструють ті самі патерни деградації. Архітектурні рішення у таких системах нерідко приймаються без формальної верифікації їх поведінки під навантаженням, що знижує обґрунтованість вибору між альтернативами і ускладнює відтворення успішних практик. Розроблення методології, що поєднує моделювання реального стану системи, емпіричне тестування та багатокритеріальне оцінювання рішень в єдиному аналітичному циклі, є актуальним завданням для спеціальності «Системний аналіз».

Питання масштабованості та надійності серверних систем досліджуються у зарубіжній науковій літературі переважно в контексті мікросервісних архітектур, патернів кешування та асинхронної обробки даних. Методологічну основу багатокритеріального оцінювання технічних рішень складає метод аналізу ієрархій Сааті, що набув поширення у задачах вибору архітектурних альтернатив. Водночас досліджень, що інтегрують UML-моделювання реального стану системи,

навантажувальне тестування та формальне порівняння запропонованих заходів у єдиній рамці, у вітчизняній літературі виявлено недостатньо.

Метою є підвищення продуктивності та масштабованості бекенд-системи шляхом виявлення архітектурних вузьких місць та оптимізації обробки запитів на основі системно-аналітичного дослідження її архітектури та характеристик продуктивності під навантаженням. Досягнення мети передбачає дослідження теоретичних засад надійності та масштабованості бекенд-систем, побудову UML-моделей процесів обробки запитів у стані As-Is та ідентифікацію структурних обмежень, проведення навантажувального тестування зі статистичною декомпозицією чинників деградації, розроблення системи моніторингу й архітектурних рішень та оцінювання їх ефективності методом аналізу ієрархій.

Об'єкт дослідження - Бекенд-система веб-платформи galaxy4games.

Предмет дослідження - методи підвищення продуктивності, надійності та масштабованості бекенд-систем за допомогою кешування, асинхронної обробки та оптимізації баз даних.

У роботі застосовано UML-моделювання для формалізації процесів обробки запитів у стані As-Is, навантажувальне тестування засобами Locust для отримання емпіричних метрик продуктивності, статистичний аналіз для декомпозиції чинників деградації, модель масового обслуговування M/M/1 для кількісного обґрунтування горизонтального масштабування та метод аналізу ієрархій для порівняльного оцінювання альтернативних архітектурних рішень.

Теоретична значущість роботи полягає у розробленні інтегрованого аналітичного циклу, що відтворюваний для дослідження інших бекенд-систем зі схожим характером навантаження. Практична значущість визначається тим, що запропонований комплекс заходів забезпечив суттєве зростання пропускної здатності системи та скорочення часу відповіді.

Інформаційна база дослідження включає наукові монографії та статті з проєктування розподілених систем, офіційну документацію використаних технологій, результати навантажувального тестування та метрики системи моніторингу, отримані в рамках практичного дослідження.

РОЗДІЛ 1.

ТЕОРЕТИЧНІ ОСНОВИ НАДІЙНОСТІ ТА МАСШТАБОВАНOSTІ БЕКЕНД-СИСТЕМ

Під час проєктування серверних застосунків однією з важливих вимог є забезпечення стабільної роботи системи при зростанні кількості користувачів та обсягу оброблюваних даних. Архітектурні рішення, які ефективно виконують свої завдання на етапі розробки або при незначному навантаженні, не завжди здатні забезпечити стабільну роботу в умовах високого навантаження. У таких випадках особливого значення набувають питання масштабованості, оптимізації доступу до даних та раціонального використання обчислювальних ресурсів.

Практика розробки веб систем демонструє, що більшість проблем із продуктивністю виникають не стільки через окремі програмні помилки, скільки через особливості архітектурної побудови системи та способи взаємодії її компонентів. Тому аналіз принципів створення бекенд-систем, методів їх оптимізації та способів оцінювання ефективності є одним з головних етапів для подальшого удосконалення будь-якої програмної платформи.

Цей розділ присвячений теоретичним аспектам проєктування сучасних бекенд-систем, вивченню особливостей мікросервісної архітектури, підходів до покращення продуктивності та масштабованості веб додатків, а також методикам навантажувального тестування, які застосовуються для аналізу їх технічних характеристик.

1.1. Особливості побудови сучасних веб-систем та бекенд-архітектур

Ефективність серверної частини веб-системи визначається не лише коректністю реалізованої логіки, а й тим, наскільки її архітектура здатна витримувати зростаюче навантаження без зменшення ефективності показників доступності та часу відповіді. Організація взаємодії між компонентами, спосіб доступу до даних і розподіл обчислювального навантаження є архітектурними рішеннями, що безпосередньо визначають межі масштабованості системи ще на етапі проектування [22, с. 23].

З архітектурної точки зору веб-система складається щонайменше з двох функціонально відокремлених рівнів: клієнтського, що відповідає за відображення даних та взаємодію з користувачем, і серверного, що реалізує бізнес-логіку, забезпечує зберігання даних та обробку вхідних запитів. Взаємодія між цими рівнями здійснюється через мережу за протоколами HTTP або HTTPS [12, с. 41]. Фундаментом такої організації є клієнт-серверна модель, за якою клієнт завжди виступає ініціатором комунікації, а сервер реагує лише на вхідні запити. Це забезпечує чіткий розподіл відповідальності та дозволяє обслуговувати різноманітні клієнти - веб-браузери, мобільні застосунки, зовнішні інтеграції - з єдиної серверної інфраструктури [13].

Серверна частина - бекенд - відповідає за обробку HTTP-запитів, реалізацію бізнес-логіки, управління даними через взаємодію з базами даних, автентифікацію користувачів, інтеграцію із зовнішніми сервісами та забезпечення журналювання і моніторингу системи [24]. Саме на цьому рівні зосереджені архітектурні рішення, що визначають продуктивність і надійність системи в цілому, - тому дослідження, представлене в даній роботі, орієнтоване передусім на аналіз серверного рівня.

З точки зору внутрішньої організації серверної частини в практиці проектування сформувалися два принципово різних підходи - монолітна та мікросервісна архітектури, - кожен з яких відповідає певному класу вимог до системи.

Монолітна архітектура передбачає реалізацію всього функціоналу застосунку у вигляді єдиного розгорнутого артефакту: обробка запитів, бізнес-логіка, доступ до даних та фонові задачі функціонують в межах одного процесу і розгортаються спільно [16, с. 34]. Такий підхід виправданий на початкових етапах розробки, коли спрощення налагодження та єдина кодова база є реальними перевагами. Проте зі зростанням системи ця пов'язаність стає обмеженням, тобто зміна окремого компонента потребує повного перерозгортання, масштабувати можна лише застосунок цілком, а не окремий його модуль, що при нерівномірному навантаженні означає надмірні витрати ресурсів [12, с. 178].

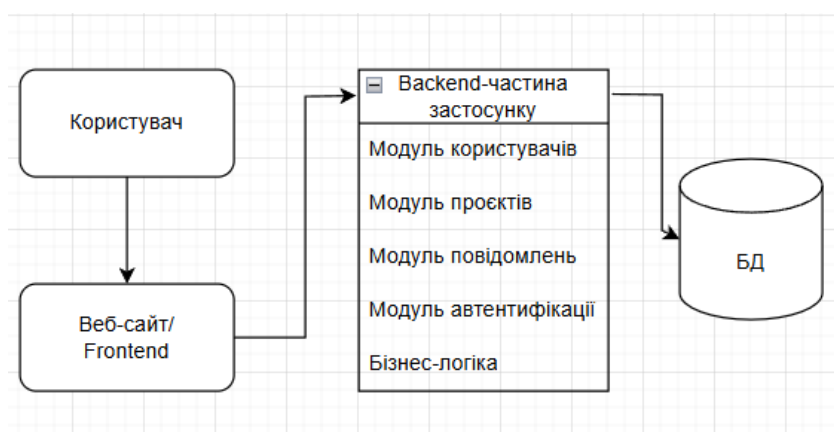


Рис. 1.1 – Монолітна архітектура бекенд-системи

На рисунку 1.1 показано типову архітектуру бекенд-системи, у якій усі функціональні модулі реалізуються в межах одного застосунку та використовують спільну базу даних.

Однак перелічені особливості монолітного підходу не лише обмежують гнучкість розгортання та масштабування системи, а й створюють додаткові труднощі під час її подальшого супроводу та розвитку. Через тісну інтеграцію компонентів зміна будь-якого модуля потенційно впливає на роботу всієї системи, що ускладнює процес тестування та підвищує ризик виникнення регресій. Крім того, масштабування монолітного застосунку здійснюється лише цілісно - неможливо збільшити ресурси лише для окремого навантаженого компонента без масштабування всієї системи, що може призводити до неефективного використання обчислювальних ресурсів. Із розширенням команди розробників також зростає

складність роботи зі спільною кодовою базою, це збільшення кількості конфліктів під час інтеграції змін, ускладнення координації процесів розробки та підтримка якості коду. Ще не менш важливим обмеженням є жорстка прив'язка до єдиного технологічного стеку, що унеможливорює вибір оптимальних технологій для реалізації окремих функціональних модулів проекту [12, с. 178].

Для подолання зазначених обмежень сформувався підхід мікросервісної архітектури, за яким система розбивається на набір незалежно розгорнутих сервісів, кожен з яких відповідає за чітко визначену функціональну область. Взаємодія між сервісами здійснюється через стандартизовані інтерфейси - як правило, HTTP REST або асинхронний обмін повідомленнями через брокери [24, с. 4]. Широке застосування цей підхід отримав після того, як технологічні компанії з масовою аудиторією - Netflix, Amazon, Uber - публічно описали перехід від монолітних рішень до розподілених архітектур як спосіб подолання стелі масштабованості.

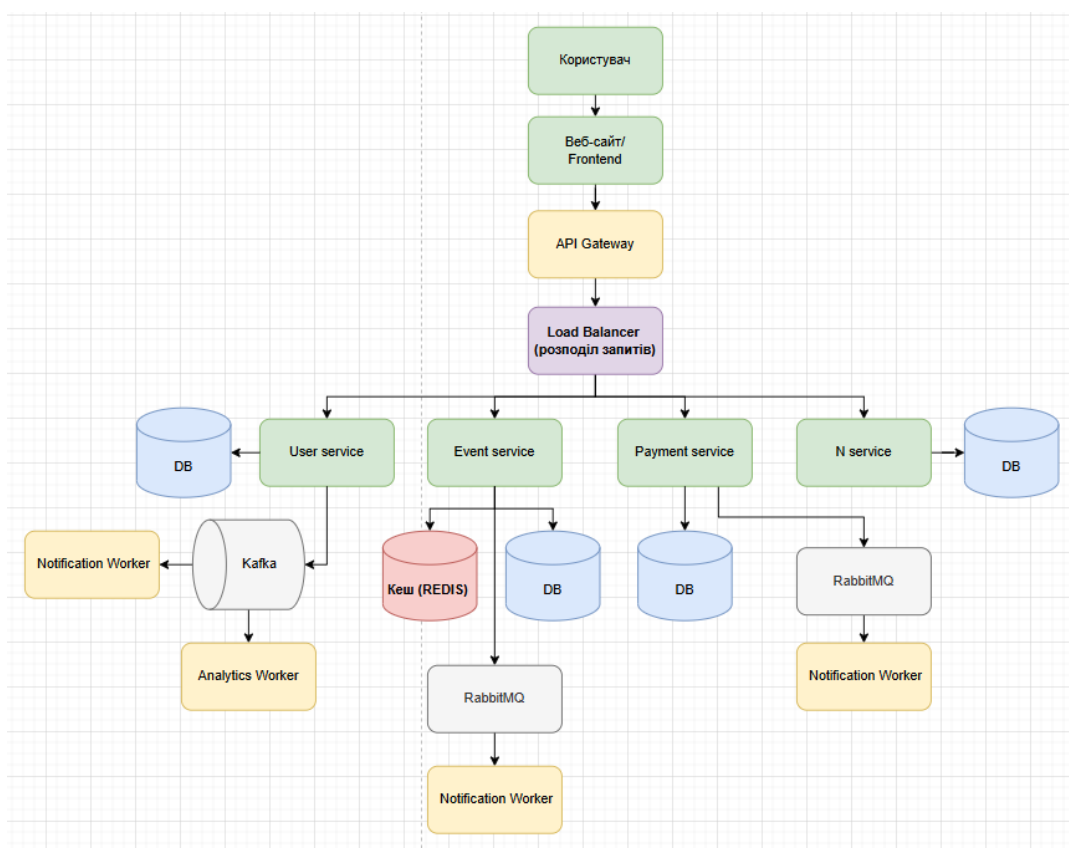


Рис. 1.3 - Схема мікросервісної архітектури бекенд-системи

На рисунку 1.3 показано типову організацію мікросервісної системи: вхідні запити надходять через шлюз API та балансувальник навантаження, розподіляються між незалежними сервісами, кожен з яких має власне ізольоване сховище даних. Асинхронна взаємодія між компонентами реалізується через брокери повідомлень, що дозволяє масштабувати окремі вузли незалежно один від одного.

Ключовою перевагою мікросервісної архітектури є незалежність розгортання: кожен сервіс може бути оновлений або масштабований без впливу на решту системи. При нерівномірному розподілі навантаження це дозволяє нарощувати ресурси лише для навантаженого компонента, а не для всього застосунку [24, с. 12]. Крім того, кожен сервіс може використовувати технологічний стек, оптимальний для своєї задачі, що підвищує гнучкість технологічних рішень.

Водночас розподілений характер архітектури привносить нові операційні виклики. Трасування помилок ускладнюється, оскільки збій в одному сервісі може проявлятися як некоректна відповідь в іншому, і встановлення першопричини вимагає аналізу декількох незалежних компонентів одночасно. Забезпечення узгодженості даних між сервісами є нетривіальною задачею через неможливість застосування класичних ACID-транзакцій у розподіленому середовищі. Мережева взаємодія між сервісами є повільнішою та менш надійною порівняно з внутрішніми викликами функцій у моноліті, що вимагає реалізації механізмів повторних спроб і таймаутів [12, с. 192].

Для систематизації відмінностей між двома підходами наведено порівняльну таблицю 1.1.

Таблиця 1.1 - Порівняльний аналіз монолітної та мікросервісної архітектур

Характеристика	Монолітна архітектура	Мікросервісна архітектура
Структура системи	Єдиний неподільний модуль	Набір незалежних сервісів
Розгортання	Цілісне, одним артефактом	Незалежне для кожного сервісу
Масштабування	Лише горизонтальне, цілісне	Гнучке, на рівні окремих сервісів
Технологічний стек	Єдиний для всього застосунку	Може відрізнятись між сервісами
Складність розробки	Низька на старті, зростає з часом	Вища на старті, стабілізується при зростанні
Налагодження та трасування	Просте, в межах одного процесу	Складне, потребує розподіленого трасування
Відмовостійкість	Збій одного компонента впливає на всю систему	Ізоляція збоїв на рівні окремого сервісу
Узгодженість даних	Класичні ACID-транзакції	Потребує патернів розподіленої узгодженості
Оптимальний розмір команди	Мала або середня команда	Середня або велика команда
Типові сценарії застосування	MVP, невеликі системи, стартапи	Великі навантажені платформи, enterprise

Джерело: складено автором на основі [12, 16, 23]

Аналіз наведених характеристик свідчить про те, що вибір між двома підходами визначається не абстрактними перевагами одного над іншим, а відповідністю архітектурного рішення конкретним вимогам: очікуваному навантаженню, вимогам до масштабованості окремих компонентів та допустимій операційній складності. Досліджувана в рамках даної роботи система побудована за мікросервісним принципом, що зумовлює специфічний характер проблем її продуктивності та надійності і визначає підходи до їх аналізу, які розглядаються в наступних розділах.

1.2. Методи забезпечення продуктивності та масштабованості бекенд-систем

Продуктивність та масштабованість є взаємопов'язаними, але різними характеристиками інформаційної системи. Продуктивність відображає здатність системи забезпечувати прийнятний час обробки запитів за певного рівня навантаження, тоді як масштабованість - здатність зберігати ці показники при зростанні кількості користувачів або інтенсивності запитів шляхом нарощування доступних ресурсів. Висока продуктивність у поточному режимі не гарантує масштабованості зі збільшенням навантаження можуть проявлятися архітектурні обмеження, що залишаються непомітними в умовах звичайної експлуатації. Оцінка масштабованості має будуватися на аналізі параметрів навантаження та способів реагування системи на їх зміну, що визначає вибір конкретних методів оптимізації, розглянутих у даному пункті [12, с. 32 - 42].

Базовим механізмом побудови горизонтально масштабованих систем є балансувальник навантаження (Load Balancer) - компонент, що виступає єдиною точкою входу для вхідних запитів і розподіляє їх між наявними екземплярами сервісу. Алгоритм розподілу обирається залежно від характеру навантаження, Round Robin забезпечує почерговий розподіл між екземплярами, Least Connections спрямовує запит до вузла з найменшою кількістю активних з'єднань, IP Hash прив'язує конкретного клієнта до конкретного екземпляра. Окрім балансування, цей компонент здійснює моніторинг стану вузлів і автоматично виключає недоступні екземпляри з пулу, що підвищує загальну відмовостійкість системи [17]. Окрім розподілу навантаження, балансувальник виконує функцію моніторингу стану екземплярів - при виявленні недоступного екземпляра він автоматично виключає його з пулу та перенаправляє трафік на справні вузли, що підвищує загальну відмовостійкість системи.

Для розподілу навантаження між операціями читання та запису на рівні бази даних застосовується патерн Read Replica. Він передбачає підтримку кількох копій основного вузла, які отримують зміни через механізм реплікації та обслуговують запити читання, тоді як основний вузол приймає лише операції запису. При типовому для веб-застосунків співвідношенні читань до записів 80:20 або 90:10 це дозволяє суттєво збільшити загальну пропускну здатність бази даних без змін у структурі даних або логіці застосунку [27, с. 186-199].

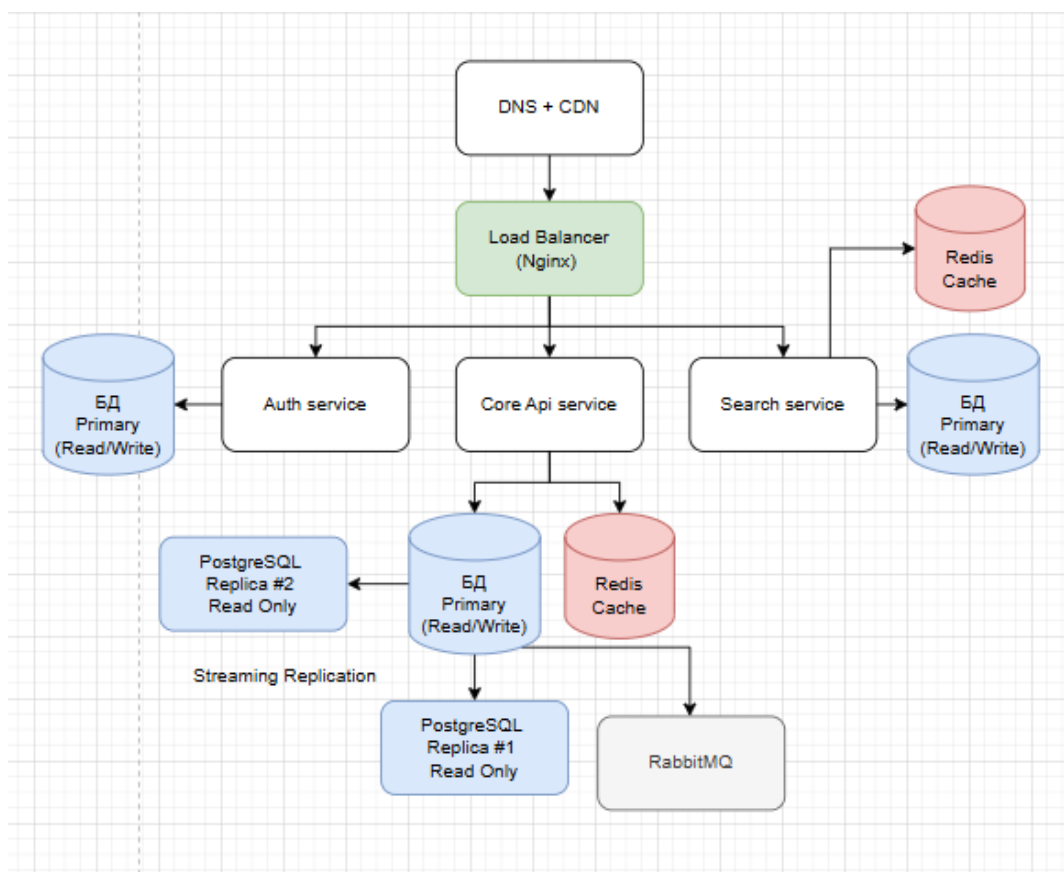


Рис 1.4 - Схема горизонтального масштабування з балансуванням навантаження та реплікацією БД

На рисунку 1.4 показано узагальнену схему горизонтально масштабованої системи: вхідні запити надходять до балансувальника навантаження, який розподіляє їх між екземплярами сервісу; операції читання спрямовуються до реплік бази даних, тоді як операції запису - до основного вузла. Такий розподіл знижує навантаження на первинний вузол і підвищує пропускну здатність підсистеми зберігання.

Зниження навантаження на базу даних досягається також через кешування даних на рівні застосунку. Одним з найбільш поширених патернів є Cache-Aside: перед зверненням до бази даних сервіс перевіряє наявність даних у кеші. У разі влучення (cache hit) результат повертається клієнту безпосередньо з кешу; у разі промаху (cache miss) сервіс звертається до бази даних, зберігає отриманий результат у кеші та повертає його клієнту. Такий підхід забезпечує поступове наповнення кешу лише тими даними, що реально запитуються [27].

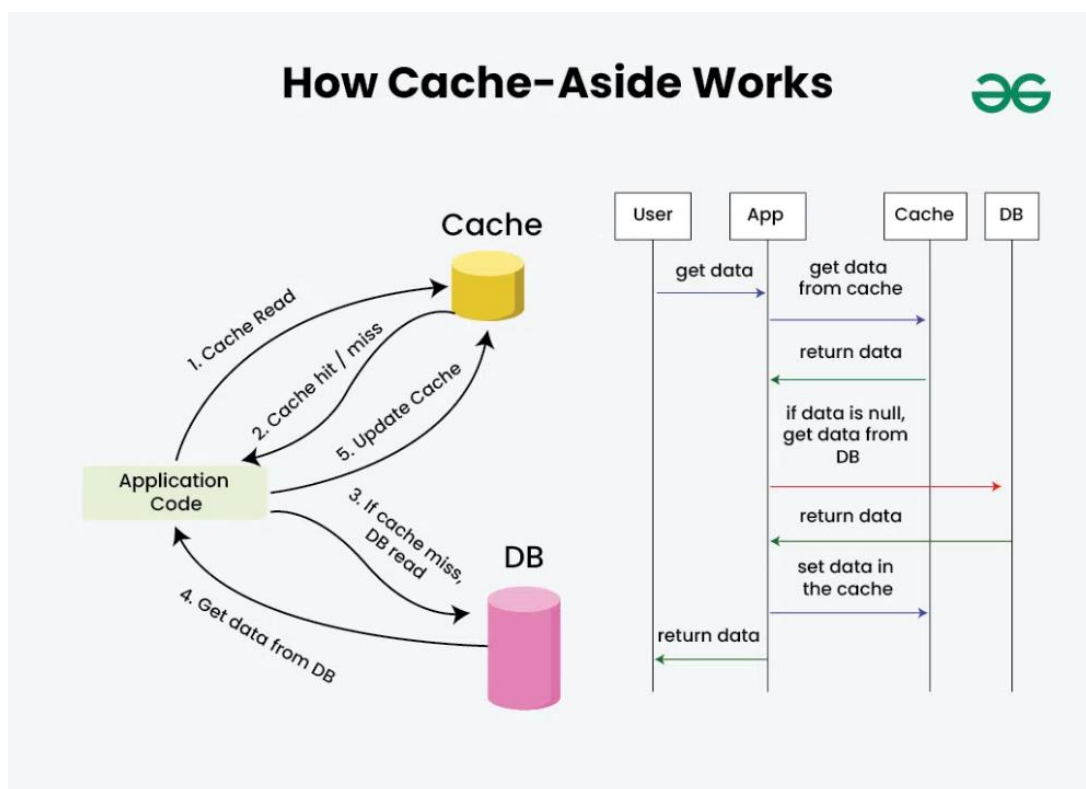


Рис. 1.5 - Алгоритм роботи патерну Cache-Aside. [8]

На рисунку 1.5 зображено принцип роботи патерна Cache-Aside, який використовується для прискорення доступу до даних шляхом застосування кешу. Спочатку застосунок виконує запит до кешу. Якщо дані знайдено (cache hit), вони одразу повертаються користувачу без звернення до бази даних. У разі відсутності даних у кеші (cache miss) застосунок звертається до бази даних, отримує необхідну інформацію, зберігає її в кеші та повертає результат користувачу.

Невід'ємною складовою стратегії кешування є параметр TTL, що визначає тривалість актуальності запису в кеші. Після його закінчення запис видаляється, і наступне звернення ініціює оновлення даних із першоджерела. Вибір значення TTL

є балансуванням між двома ризиками: завищене значення підвищує ймовірність повернення застарілих даних, занижене - знижує ефективність кешування та збільшує навантаження на базу даних [15]. Для даних з низькою частотою змін - наприклад, довідникових записів або результатів агрегованих запитів - типові значення TTL становлять від кількох хвилин до кількох годин.

Ефективність кешування кількісно характеризується показником Cache Hit Ratio (CHR) - часткою запитів, що були оброблені безпосередньо з кешу без звернення до бази даних. Показник обчислюється за формулою:

(1.1)

$$CHR = \frac{Cache\ Hits}{Cache\ Hits + Cache\ Misses}$$

Де Cache Hits - кількість успішних звернень до кешу;

Cache Misses - кількість неуспішних звернень до кешу, що потребували отримання даних із бази даних[5].

Значення CHR належить інтервалу [0; 1]. Його зростання свідчить про підвищення ефективності використання кешу і безпосередньо впливає на зниження навантаження на базу даних, скорочення затримок та підвищення масштабованості системи при зростанні кількості одночасних користувачів.

Асинхронна обробка задач є підходом, за якого взаємодія між компонентами системи здійснюється через обмін повідомленнями без необхідності негайного отримання відповіді. На відміну від синхронної моделі, де компонент блокується в очікуванні завершення зовнішньої операції, асинхронна взаємодія дозволяє сервісам працювати незалежно, що знижує зв'язність, підвищує відмовостійкість і спрощує горизонтальне масштабування окремих компонентів [26, с. 29 - 30].

Показовим прикладом операції, що виграє від асинхронного виконання, є надсилання електронної пошти - задача, обмежена продуктивністю введення-виведення, виконання якої залежить від взаємодії із зовнішнім поштовим сервісом. У синхронній моделі застосунок блокується до завершення цієї операції, що

безпосередньо погіршує час відповіді при одночасному надходженні великої кількості запитів. Перенесення таких операцій у фоновий режим через чергу повідомлень звільняє основний потік виконання та підвищує пропускну здатність системи [7, с. 22 - 24]. Саме цей сценарій є актуальним для досліджуваної системи і детально розглядається у третьому розділі.

Оптимізація роботи з базою даних є одним із ключових напрямів підвищення продуктивності бекенд-систем, оскільки база даних часто виступає обмежувальним фактором під час обробки великої кількості запитів. Окремим напрямом оптимізації є управління підключеннями до бази даних через пул з'єднань. Встановлення нового з'єднання супроводжується накладними витратами на мережеву ініціалізацію та автентифікацію, тому пул підтримує набір попередньо створених з'єднань, що повторно використовуються різними запитами. Розмір пулу потребує балансування: недостатня кількість з'єднань призводить до черг очікування, надмірна - до надлишкового споживання пам'яті на сервері бази даних і зниження її загальної продуктивності [6].

Доповнює цей механізм індексування таблиць. Індекс - допоміжна структура даних, що дозволяє СУБД знаходити необхідні записи без повного перегляду таблиці. Найпоширенішою індексною структурою є B^+ -дерево, яке зберігає ефективність пошуку за умови постійних вставок і видалень. Проте підтримка індексів створює додаткові витрати під час операцій запису та потребує додаткового дискового простору, тому їх застосування виправдане лише для полів, що активно використовуються у предикатах запитів або умовах сортування [25, с. 634].

Окрім методів масштабування, у розподілених системах застосовують патерни захисту від перевантажень і каскадних відмов. Обмеження конкурентності (за допомогою семафора або патерну Bulkhead) встановлює верхню межу на кількість одночасно виконуваних ресурсомістких операцій, запобігаючи вичерпанню процесорного часу та «голодуванню» циклу подій. Патерн «запобіжник» (Circuit Breaker) відстежує частоту помилок при зверненні до зовнішнього компонента і, перевищивши поріг, тимчасово припиняє виклики, повертаючи відмову миттєво (стратегія Fail Fast) замість очікування таймауту. Це

вивільняє ресурси, дає несправному компоненту відновитися, а клієнту - застосувати повторні спроби зі зростаючою затримкою (Exponential Backoff). [12, с. 634].

Розглянуті методи у промислових системах реалізуються не ізольовано, а як взаємодоповнювальний комплекс заходів. Горизонтальне масштабування без кешування лише переносить вузьке місце з рівня сервісу на рівень бази даних; кешування без асинхронної обробки не усуває блокування, що виникає при виконанні тривалих зовнішніх операцій. Саме тому практичне дослідження, представлене в даній роботі, розглядає ці підходи у взаємодії, а не як незалежні альтернативи.

1.3. Методи оцінювання продуктивності інформаційних систем

Оцінювання продуктивності інформаційної системи є необхідною умовою обґрунтованого вибору між альтернативними архітектурними рішеннями: без кількісних даних про поведінку системи під навантаженням будь-які висновки щодо доцільності оптимізаційних заходів залишаються декларативними. Для цього застосовуються метрики, що характеризують швидкість обробки запитів, рівень використання ресурсів і здатність системи утримувати задані показники при зростанні навантаження [18]. У даному пункті розглянуто основні метрики продуктивності, типологію навантажувального тестування та обґрунтовано вибір інструментарію для практичного дослідження.

Центральною метрикою продуктивності веб-систем є час відповіді (Response Time) - інтервал між моментом відправлення клієнтом запиту та моментом отримання повної відповіді від сервера. Ця метрика безпосередньо визначає якість користувацького досвіду і є першим індикатором зменшення ефективності при зростанні навантаження[30]. Для агрегованої оцінки за сукупністю вимірювань використовується середнє арифметичне:

(1.2)

$$\bar{t} = \frac{1}{n} \sum_{i=1}^n t_i$$

де $\bar{t}$ - середній час відповіді, мс;

t_i - час відповіді окремого i -го запиту, мс;

n - загальна кількість запитів у вибірці.

Проте середнє значення може приховувати реальний характер розподілу. Кілька аномально тривалих запитів суттєво зміщують його вгору, тоді як при рівномірному розподілі воно недооцінює частоту повільних відповідей. Тому в практиці аналізу продуктивності поряд із середнім арифметичним

використовуються перцентилі - зокрема p95 та p99. Значення p95 означає, що 95% запитів було оброблено за час, що не перевищує це значення; лише 5% виконувались повільніше. Цей показник є більш репрезентативним, оскільки відображає поведінку системи для переважної більшості користувачів, не згладжуючи проблеми за рахунок масиву швидких запитів [19]. Для виявлення граничних сценаріїв обслуговування додатково аналізується p99 та p99.9.

Затримка (Latency) є суміжним, але відмінним поняттям. Якщо час відповіді охоплює повний цикл від відправлення запиту до отримання відповіді, то латентність характеризує затримку на конкретній ділянці - наприклад, час очікування вільного з'єднання з базою даних або час перебування запиту в черзі виконання. Розрізнення цих понять є принциповим при локалізації вузьких місць: висока загальна латентність при прийнятному часі відповіді вказує на затримку всередині конкретного компонента, а не в системі загалом [10, с. 18 - 30].

Пропускна здатність (Throughput) характеризує кількість запитів, що система здатна обробити за одиницю часу, і вимірюється в запитах за секунду (RPS). Між пропускнуою здатністю і часом відповіді існує нелінійна залежність при наближенні навантаження до межі пропускнуої здатності час відповіді починає стрімко зростати через накопичення черги, навіть якщо обробка окремого запиту не сповільнилась [12, с. 36 - 37]. Виявлення цієї точки перегину є однією з ключових цілей навантажувального тестування.

Відсоток помилок (Error Rate) відображає частку запитів, що завершилися помилкою - HTTP-кодом 5xx на боці сервера або таймаутом на боці клієнта. Аналітично важливо розрізняти два сценарії зменшення ефективності - це те що система може повертати прийнятний відсоток помилок при неприйнятно великому часі відповіді, або навпаки - відповідати помилками швидко при перевантаженні. Обидва сценарії є проблематичними, однак мають різні першопричини і вимагають різних рішень [28].

Таблиця 1.2 - Основні метрики оцінювання продуктивності веб-систем

Метрика	Одиниця виміру	Що характеризує	Типове цільове значення
Response Time (середній)	мс	Швидкість відповіді	< 200-500 мс
p95	мс	Досвід більшості користувачів	< 1 000 мс
p99	мс	Граничні сценарії обслуговування	< 2 000 мс
Throughput (RPS)	запитів/сек	Загальна потужність системи	залежить від вимог
Latency	мс	Затримка на окремій ділянці	< 50 мс
Error Rate	%	Надійність під навантаженням	< 0.1%

Джерело: складено автором на основі [10, 28]

Для вимірювання наведених метрик в умовах, наближених до реальних, застосовується навантажувальне тестування - методологія, що передбачає імітацію одночасної роботи великої кількості користувачів і спостереження за поведінкою системи при контрольованому зростанні навантаження. Ключовою метою є не просто підтвердження факту працездатності системи, а виявлення точки зменшення ефективності - рівня навантаження, при якому показники виходять за допустимі межі [4].

Залежно від мети розрізняють три основних види навантажувальних тестів. Load Testing передбачає поступове збільшення кількості одночасних користувачів до очікуваного робочого рівня з утриманням цього рівня впродовж певного часу - для перевірки стабільності системи при штатному навантаженні. Stress Testing виходить за межі очікуваного навантаження з метою визначення абсолютної межі пропускної здатності та характеру зменшення ефективності: поступової чи раптової. Spike Testing моделює різкі короточасні зростання трафіку і оцінює здатність системи не лише витримувати пікове навантаження, а й відновлюватись після нього [4].

Таблиця 1.3 - Порівняльна характеристика видів навантажувального тестування

Вид тестування	Характер навантаження	Мета	Результат
Load Testing	Поступове зростання до робочого рівня	Перевірка стабільності при штатному навантаженні	Метрики при очікуваному навантаженні
Stress Testing	Навантаження понад робочий рівень	Визначення межі пропускну здатності	Точка зменшення ефективності системи
Spike Testing	Різкий короточасний стрибок	Перевірка реакції на пікові навантаження	Час відновлення після піку

Джерело: розроблено автором

Для практичного проведення навантажувального тестування в рамках даного дослідження обрано інструмент Locust. Вибір обумовлений двома факторами. По-перше, сценарії навантаження описуються засобами Python без окремої мови специфікацій, що дозволяє реалізовувати складну поведінку користувачів - умовні переходи, зважену імовірність виконання окремих дій, параметризовані запити. По-друге, асинхронна модель виконання на основі `gevent` забезпечує достатню масштабованість для імітації сотень одночасних з'єднань в умовах одного тестового вузла без залучення розподіленої інфраструктури [14]. Вбудований веб-інтерфейс Locust відображає зміну метрик у реальному часі, що дозволяє фіксувати момент настання зменшення ефективності безпосередньо під час тесту.

У контексті спеціальності аналізу отримані в результаті тестування метрики виступають не самоціллю, а вхідними даними для подальшого впровадження альтернатив у третьому розділі вони використовуються як критеріальна база для порівняльного оцінювання запропонованих архітектурних рішень методом аналітичної ієрархії. Практичне застосування описаної методології розглянуто у пункті 2.2.

РОЗДІЛ 2.

АНАЛІЗ ТА МОДЕЛЮВАННЯ ХАРАКТЕРИСТИК БЕКЕНД-СИСТЕМИ ВЕБ-ПЛАТФОРМИ GALAXY4GAMES

У першому розділі було досліджено теоретико-методологічні засади забезпечення надійності та масштабованості розподілених програмних комплексів, а також проаналізовано сучасні підходи до оцінювання їх продуктивності під навантаженням. Проте для розроблення ефективних практичних рекомендацій та архітектурних трансформацій необхідне глибоке прикладне дослідження конкретної програмної системи, що функціонує в реальних експлуатаційних умовах.

Метою другого розділу є комплексний системний аналіз, формалізація архітектури та експериментальне оцінювання характеристик надійності й масштабованості бекенд-системи компанії. Реалізація окресленої мети передбачає послідовне вирішення таких дослідницьких завдань, як аналіз організаційно-технічного контексту функціонування платформи та ідентифікацію актуальних технологічних викликів, побудову структурно-функціональних UML-моделей процесів обробки запитів у стані «як є» (As-Is); проведення комплексного навантажувального тестування для отримання емпіричних метрик продуктивності, а також статистичну декомпозицію виявлених чинників зменшення ефективності надійності системи.

2.1. Дослідження об'єкта та UML-моделювання процесів обробки запитів

Об'єктом практичного дослідження в рамках даної роботи виступає серверна частина веб-платформи ІТ-компанії, що спеціалізується на розробці казуальних ігор для мобільних пристроїв та персональних комп'ютерів. Компанія має понад 15 років досвіду у створенні ігрових продуктів і за цей час сформувала розгалужену інфраструктуру програмних сервісів, які забезпечують функціонування як внутрішніх продуктів, так і публічних веб-ресурсів компанії. Веб-платформа, що досліджується в цьому розділі, виконує функцію корпоративного портфоліо - вона презентує реалізовані та перспективні проєкти потенційним клієнтам і партнерам. Попри відносно вузьку призначеність, серверна частина платформи обслуговує повний цикл API-запитів від клієнтської частини сайту, а точніше отримання каталогу проєктів до обробки форм зворотного зв'язку з подальшим надсиланням електронних повідомлень. Це означає, що навіть за відсутності складної ігрової логіки система повинна витримувати конкурентне навантаження, характерне для публічних веб-ресурсів із непередбачуваними піками трафіку.

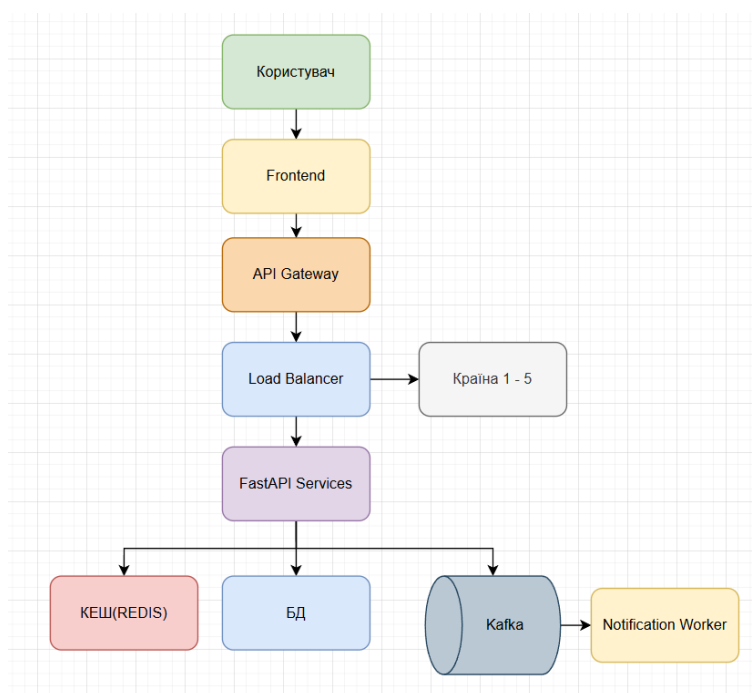


Рис. 2.1 - Архітектура серверної частини веб-платформи компанії.

На рисунку 2.1 відображено архітектуру серверної частини веб-платформи до впровадження оптимізацій: основні компоненти системи та зв'язки між ними.. Архітектура серверної частини реалізована на основі мікросервісного підходу та включає компоненти для масштабування й розподілу навантаження FastAPI services, як рівень прикладної логіки, PostgreSQL як основне сховище даних, Redis для кешування та Apache Kafka для організації асинхронної взаємодії між сервісами. Присутність Redis, Kafka та балансувальника навантаження свідчить про намір забезпечити горизонтальне масштабування та зменшення часу відповіді системи. Проте аналіз фактичної конфігурації показав, що частина цих механізмів використовувалась обмежено наприклад, кешування охоплювало лише окремі категорії даних, асинхронна взаємодія не застосовувалась для всіх тривалих операцій, а балансувальник обслуговував один серверний екземпляр. У результаті фактична поведінка системи відрізнялась від очікуваної моделі горизонтального масштабування та створювала ризик появи вузьких місць при зростанні інтенсивності запитів.

Заумови помірного навантаження така архітектура забезпечувала стабільне функціонування веб-платформи. Проте зі збільшенням інтенсивності надходження запитів почали проявлятися характерні ознаки зменшення ефективності продуктивності, а саме, збільшення часу відповіді API, під навантаженням було зафіксовано нелінійне збільшення часу очікування доступу до бази даних та поступове накопичення одночасно активних запитів. У результаті окремі ендпоінти, що в нормальному режимі виконувались за десятки мілісекунд, демонстрували кратне збільшення затримки та епізодичні перевищення допустимого часу відповіді.

Таким чином, у межах досліджуваної системи, спостерігається суперечність між закладеними механізмами масштабування та фактичною поведінкою архітектури під навантаженням. Виявлені особливості узгоджуються з підходами до аналізу масштабованості, розглянутими у пункті 1.2, де було показано, що ефективність горизонтального масштабування визначається не лише наявністю відповідних компонентів, а і способом їх фактичного використання[3]. Для

перевірки цього припущення у наступних підрозділах виконується формалізація процесів обробки запитів засобами UML-моделювання та кількісне оцінювання впливу виявлених архітектурних обмежень за результатами навантажувального тестування.

Для формалізації припущень та виявлення структурних особливостей обробки запитів до моменту оптимізації застосовано UML-моделювання. У контексті даного дослідження воно слугує не інструментом проєктування нової системи, а засобом побудови моделі As-Is - формального опису реального стану системи, що передує будь-яким архітектурним втручанням.

Для комплексного опису досліджуваної системи побудовано два типи UML - діаграм, що взаємодоповнюють одна одну: діаграму послідовності (Sequence Diagram) та діаграму діяльності (Activity Diagram)[1]. Перша акцентує увагу на хронологічному порядку взаємодії між компонентами, друга деталізує внутрішній алгоритм обробки запиту всередині серверного компонента - з умовними розгалуженнями та точками прийняття рішень, які не відображаються на діаграмі послідовності.

Діаграма послідовності відображає повний шлях клієнтського запиту від моменту його формування у браузері користувача до отримання відповіді. Взаємодія відбувається між чотирма учасниками - користувачем, клієнтським застосунком (Frontend), серверним API та реляційною базою даних PostgreSQL. Суцільними стрілками позначено виклики та запити, пунктирними - повернення результатів. Смуга активації на учаснику «Серверний API» відображає проміжок часу, протягом якого сервер утримує з'єднання відкритим в очікуванні відповіді від бази даних.

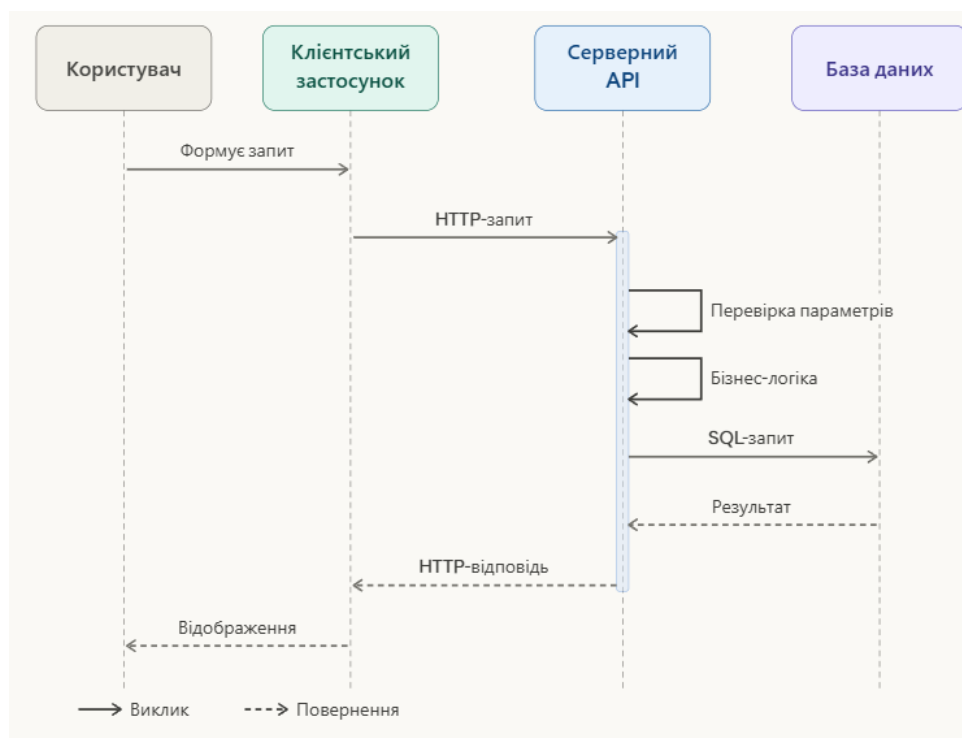


Рис. 2.2 - Діаграма послідовності обробки клієнтського запиту

Аналіз діаграми на рисунку 2.2 виявляє ключову структурну особливість системи у її поточному стані: при кожному зверненні до ендпоінту серверний API виконує повний SQL-запит до бази даних і лише після отримання результату формує відповідь клієнту. Таким чином, кожен клієнтський запит породжує окреме звернення до PostgreSQL незалежно від того, чи запитувались аналогічні дані раніше. При збільшенні кількості одночасних користувачів це призводить до накопичення черги SQL-запитів та зростання навантаження на базу даних - перша з трьох гіпотез, сформульованих за результатами аналізу архітектури в підрозділі 2.1, отримує структурне підтвердження на рівні моделі.

Особливу увагу привертає смуга активації серверного API, яка охоплює всі етапи обробки запиту: перевірку параметрів, виконання бізнес-логіки, очікування результату від бази даних та формування відповіді. Це свідчить про синхронний характер роботи системи. Сервер утримує робочий потік та мережеве з'єднання відкритими до завершення всіх операцій, що за умов високого навантаження може призводити до вичерпання доступних ресурсів та зменшення ефективності продуктивності.

Для деталізації внутрішньої логіки обробки запиту на рівні серверного API побудований відповідний алгоритм.

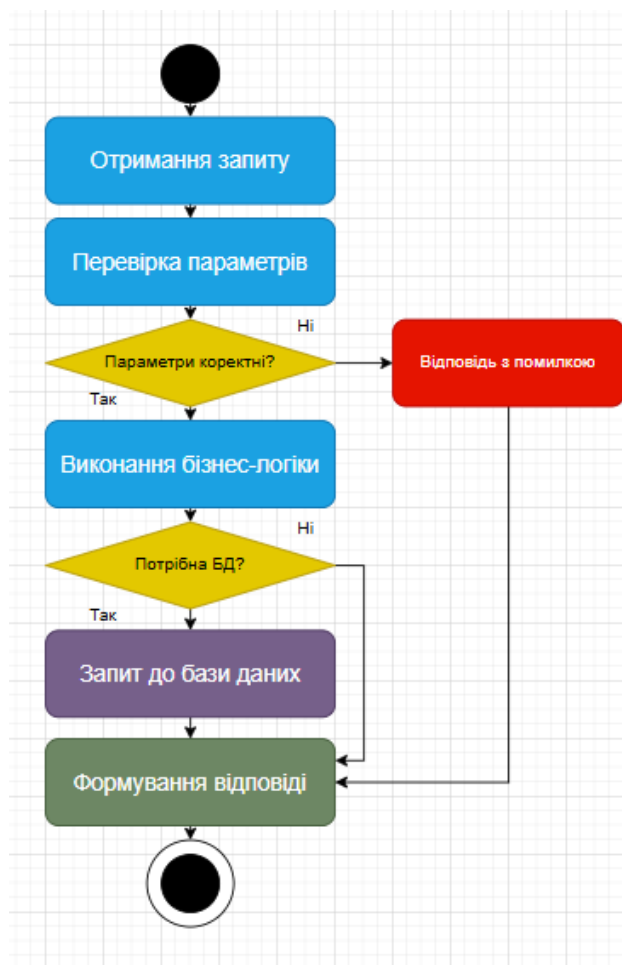


Рис. 2.3 - Алгоритм обробки запитів

На рисунку 2.3 відображено основні кроки обробки запиту, отримання HTTP-запиту, перевірка коректності вхідних параметрів, виконання бізнес-логіки, звернення до бази даних та формування відповіді клієнту. Потік обробки розгалужується у двох точках прийняття рішень. У першій перевіряється коректність вхідних параметрів, у разі виявлення помилок система негайно формує відповідь з відповідним кодом, оминаючи виконання бізнес-логіки - рання валідація скорочує час обробки некоректних запитів і знижує навантаження на внутрішні компоненти. У другій точці визначається необхідність звернення до бази даних, якщо виконання бізнес-логіки потребує отримання або модифікації даних, система формує SQL-запит до PostgreSQL і після отримання результату повертає відповідь клієнту.

Принципово важливим є те, що обидві гілки потоку завершуються синхронним поверненням відповіді - жодна зі стадій не виконується паралельно або у фоновому режимі. Зокрема, ендпоінт POST /api/contact, що обробляє форму зворотного зв'язку, виконував валідацію, запис до бази даних та надсилання електронного листа через зовнішній SMTP-сервіс послідовно в межах одного потоку - сервер утримував з'єднання відкритим до отримання підтвердження від поштового сервісу, що додавало від 1 до 3 секунд до часу відповіді на кожен запит незалежно від рівня навантаження. Це структурно підтверджує третю гіпотезу щодо фіксованої затримки від синхронного виклику SMTP.

Порівняльний аналіз двох побудованих моделей дозволяє сформулювати центральний структурний висновок: продуктивність досліджуваної системи лінійно залежить від доступності та швидкодії бази даних, оскільки жоден з етапів обробки запиту не виконується паралельно або у фоновому режимі. Діаграма послідовності показує, що найбільша частина часу обробки зосереджена у фазі очікування відповіді від PostgreSQL, діаграма діяльності підтверджує, що алгоритм є повністю лінійним і синхронним на рівні окремого сервісу. Для кількісного підтвердження впливу цього архітектурного обмеження на продуктивність у наступному пункті проведено навантажувальне тестування.

2.2. Навантажувальне тестування та виявлення «вузьких місць» у системі

Побудовані у пункті 2.1 UML-моделі дозволили формалізувати три гіпотези щодо архітектурних обмежень системи, вичерпання пулу з'єднань до бази даних при конкурентному навантаженні, надмірне навантаження на PostgreSQL через відсутність кешування частотних запитів, та фіксована затримка від синхронного виклику SMTP у межах основного потоку. Для кількісної перевірки цих гіпотез проведено навантажувальне тестування серверної частини платформи з використанням інструменту Locust. На відміну від статичного аналізу коду, навантажувальне тестування відтворює реальні умови експлуатації і виявляє вузькі місця, що проявляються лише при конкурентному навантаженні - тобто саме ті обмеження, що були передбачені сформульованими гіпотезами. Продуктивність розподіленої системи слід оцінювати не в умовах одиночних запитів, а саме при конкурентному навантаженні, оскільки більшість проблем масштабованості є латентними і проявляються лише при певному пороговому значенні кількості одночасних сесій [12].

За класифікацією, наведеною у пункті 1.3, реалізований сценарій відповідає Load Testing з елементами Stress Testing: поступове нарощування навантаження дозволяє зафіксувати момент зменшення ефективності та визначити граничну пропускну здатність системи. Для тестування визначено три API-ендпоінти, що охоплюють принципово різні патерни навантаження. GET /api/projects - ендпоінт отримання каталогу проєктів, найбільш частотний у системі, оскільки викликається з головної та кількох внутрішніх сторінок. GET /api/projects/{id} - отримання детальної інформації про окремий проєкт за ідентифікатором. POST /api/contact - обробка форми зворотного зв'язку, на відміну від попередніх, виконує не лише читання, а й запис до бази даних із подальшим синхронним надсиланням електронного листа через зовнішній SMTP-сервіс.

Фрагмент реалізованого сценарію навантажувального тестування наведено в додатку А.

Сценарій передбачав поступове збільшення кількості одночасних віртуальних користувачів до 4 000 із темпом підключення 100 користувачів на секунду. Поступове наростання навантаження, на відміну від миттєвого стрибка, дозволяє спостерігати динаміку зменшення ефективності і фіксувати порогові значення навантаження, при яких починають проявлятися проблеми продуктивності. Для кількісної оцінки використовувались чотири метрики: середній час відповіді (Average Response Time, мс), 95-й перцентиль (p95, мс), пропускна здатність (RPS) та відсоток помилок (Failure Rate, %).

Результати первинного тестування виявились показовими. При досягненні рівня 4 000 одночасних користувачів система продемонструвала критичне зростання часу відповіді на всіх трьох досліджуваних ендпоінтах. Зведені показники наведено в таблиці 2.1.

Таблиця 2.1 - Результати навантажувального тестування до оптимізації

Ендпоінт	Кількість запитів	Медіана, мс	Середній час відповіді, мс	p95, мс	p99, мс	Min, мс	Max, мс	RPS
POST /api/contact	3 791	12 000	11 939	15 000	15 000	9 010	15 276	44.2
GET /api/projects	14 202	12 000	11 577	16 000	17 000	7 011	17 463	145.4
GET /api/projects/1	2 153	11 000	10 820	15 000	15 000	7 012	15 043	27.4
GET /api/projects/2	2 106	11 000	11 006	15 000	15 000	7 018	15 149	21.6
GET /api/projects/3	2 108	11 000	10 902	15 000	15 000	7 033	15 253	22.7
GET /api/projects/4	2 130	11 000	10 956	15 000	15 000	7 011	15 059	25.5
GET /api/projects/5	2 104	11 000	10 988	15 000	15 000	7 025	15 043	22.8
GET /api/projects/6	2 070	11 000	10 883	15 000	15 000	7 007	15 095	22.4
Aggregated	30 664	11 000	11 353	15 000	17 000	7 007	17 463	332

Джерело: складено автором за результатами навантажувального тестування у Locust

Отримані дані свідчать про системну деградацію продуктивності при конкурентному навантаженні. Показовим є порівняння ендпоінтів за мінімальним часом відповіді, GET-ендпоінти демонструють мінімум у діапазоні 7 007 - 7 033 мс,

тоді як POST /api/contact - 9 010 мс. Оскільки мінімальне значення відображає найкращий можливий сценарій обробки запиту, різниця у 2 секунди між GET і POST вказує на фіксовану затримку, не залежну від рівня навантаження, а саме на синхронне очікування відповіді від SMTP - сервісу. Ендпоінт GET /api/projects, що є найчастотнішим (14 202 запити - 46% від загальної кількості), показав середній час відповіді 11 577 мс - у 200 - 300 разів більше за прийнятний рівень для операції читання відносно статичних даних. За даними дослідження Google Think, проведеного на вибірці понад 900 тис. мобільних сесій, при часі завантаження сторінки понад 3 секунди більше половини мобільних користувачів залишають сайт [9] - тобто фактичні затримки системи перевищують критичний поріг у 3 - 5 разів.

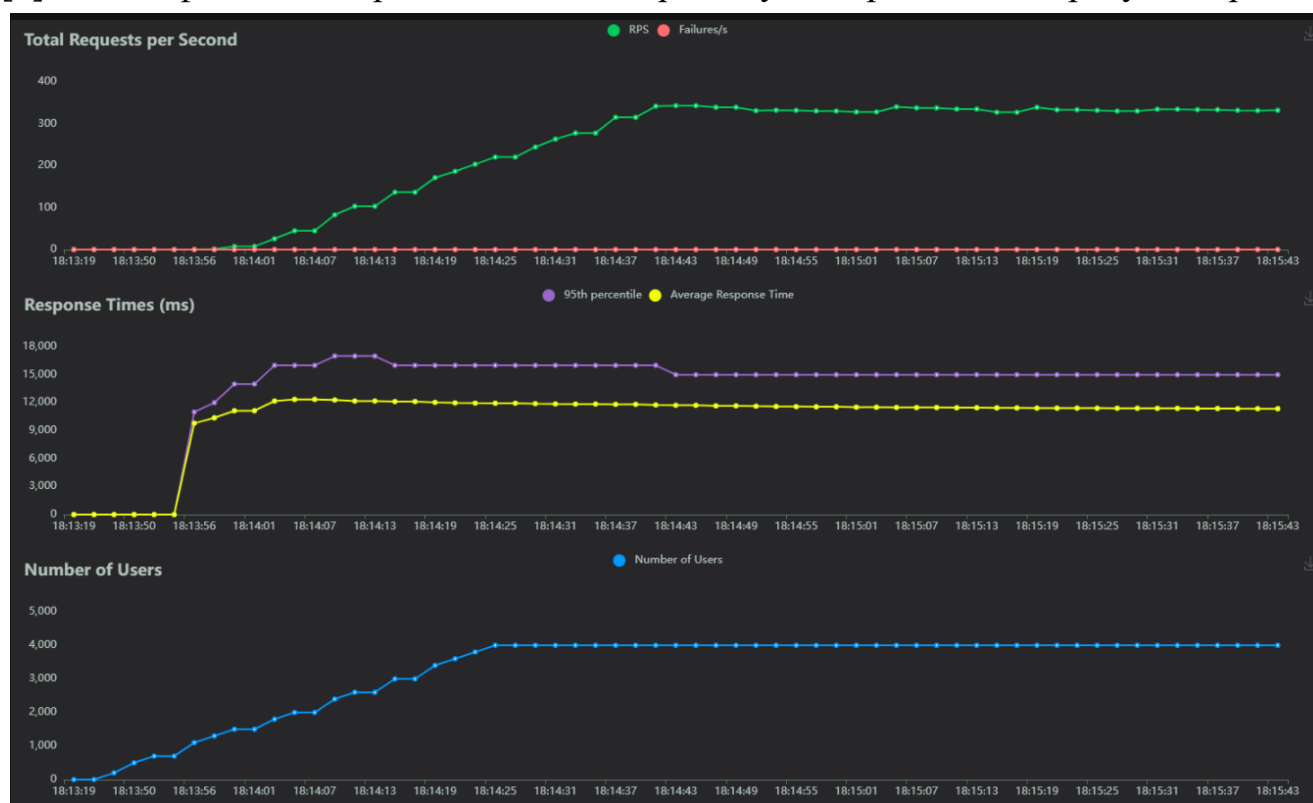


Рис. 2.4 - Графіки часу відповіді та пропускної здатності системи до оптимізації

Динаміку поведінки системи під навантаженням відображають графіки, отримані в інтерфейсі Locust (рис. 2.4). На графіку Total Requests per Second виділяються дві фази, на початковому етапі пропускна здатність зростала пропорційно кількості користувачів, однак після досягнення 332 RPS зупинилась на плато - подальше збільшення навантаження до 4 000 користувачів не призвело

до зростання кількості оброблених запитів. Це є класичною ознакою насичення системи, черга запитів накопичується швидше, ніж система здатна її обробляти. Графік Response Times підтверджує, що середній час відповіді стабілізувався на рівні 10 000 - 12 000 мс, p95 - на рівні 14 000 - 15 000 мс. Відсутність HTTP-помилки при цьому свідчить не про стабільність, а про те, що система продовжувала обробляти запити - але з критичними затримками, що є характерним проявом вичерпання пулу з'єднань, а не збоєм компонентів.

Таким чином, результати навантажувального тестування кількісно підтвердили всі три гіпотези, сформульовані за результатами UML-моделювання: система досягла межі пропускної здатності на рівні 332 RPS, мінімальні значення часу відповіді вказують на системну чергу до бази даних та фіксовану затримку SMTP, а розподіл запитів за ендпоінтами демонструє непропорційне навантаження від відсутності кешування. Детальна декомпозиція виявлених чинників зменшення ефективності наведена у пункті 2.3.

2.3. Статистичний аналіз та декомпозиція чинників зменшення надійності

Результати навантажувального тестування, отримані у пункті 2.2, зафіксували факт системної зменшення ефективності продуктивності, однак самі по собі не розкривають причинно-наслідкових зв'язків між архітектурними рішеннями та спостережуваними затримками. Для переходу від констатації симптомів до ідентифікації першопричин необхідний структурований статистичний аналіз зібраних метрик у зіставленні з архітектурними моделями, побудованими у пункті 2.1.

Аналіз результатів тестування показав, що зменшення продуктивності мала системний характер і проявлялася на всіх досліджуваних ендпоінтах незалежно від типу виконуваної операції. Середній час відповіді для операцій читання та запису перевищував 10 секунд, а значення 95-го перцентилля досягало 15 - 16 секунд. При цьому система продовжувала обробляти запити без критичних помилок, однак робила це зі значними затримками.

Першим кроком аналізу є групування зібраних метрик за типом операції - читання та запис - оскільки ці два класи мають принципово різну природу навантаження на серверну інфраструктуру. Результати групування наведено в таблиці 2.2.

Таблиця 2.2 - Порівняльний аналіз метрик продуктивності за типом операції.

Тип операції	Ендпоінт	Середній час відповіді, мс	p95, мс	Min, мс	Max, мс	Розмах (Max–Min), мс	CV, %
Читання (GET)	/api/projects	11 577	16 000	7 011	17 463	10 452	~9.0
Читання (GET)	/api/projects/{id}	~10 926	15 000	7 007	15 253	8 246	~6.5
Запис + зовн. сервіс (POST)	/api/contact	11 939	15 000	9 010	15 276	6 266	~5.2

Джерело: розроблено автором за результатами навантажувального тестування

Дані таблиці 2.2 виявляють ключову аналітичну закономірність. Ендпоінти читання GET /api/projects/{id} виконують концептуально просту операцію - отримання одного запису за первинним ключем - і в умовах нормального функціонування при наявності індексу мали б відповідати за одиниці мілісекунд. Натомість їх середній час відповіді (~10 926 мс) є порівнянним із POST /api/contact (11 939 мс), який додатково виконує запис до бази даних та виклик зовнішнього SMTP-сервісу. Збіжність показників двох принципово різних за складністю операцій свідчить про те, що вузьке місце знаходиться не на рівні складності SQL-запитів, а на рівні доступу до ресурсів бази даних як такої, система не може своєчасно отримати з'єднання незалежно від того, який запит потрібно виконати.

Додаткову аналітичну цінність має порівняння коефіцієнтів варіації. POST /api/contact демонструє найнижчий CV (~5.2%) попри найвищий мінімальний час відповіді (9 010 мс) - це означає, що затримка для цього ендпоінту є стабільною і передбачуваною, тобто фіксованою за природою. GET /api/projects, навпаки, має найвищий CV (~9.0%) та найбільший розмах (10 452 мс) - його затримка залежить від поточного рівня навантаження і зростає непропорційно зі збільшенням черги запитів до бази даних. Ця різниця у характері нестабільності є діагностичною: вона вказує на два різні механізми зменшення ефективності, що діють одночасно.

Показовим є також аналіз мінімальних значень часу відповіді як індикатора базової затримки системи за відсутності конкурентного навантаження. Мінімальний час для GET-ендпоінтів становить 7 007 - 7 033 мс, тоді як для POST /api/contact - 9 010 мс. Різниця у ~2 секунди між двома класами операцій у найкращому сценарії прямо вказує на фіксовану затримку, не пов'язану з рівнем навантаження, - а саме на синхронне очікування відповіді від зовнішнього SMTP-сервісу в межах основного потоку обробки запиту.

До архітектурних обмежень системи у стані As-Is належить також відсутність механізмів захисту від перевантажень - обмеження конкурентного виконання та запобіжника на виклики до зовнішніх сервісів, через що відмова зовнішнього компонента здатна спричинити каскадне накопичення запитів.

На підставі статистичного аналізу зібраних метрик та їх зіставлення з UML-моделями підрозділу 2.1 ідентифіковано три незалежні чинники зменшення ефективності надійності системи.

Перший чинник - вичерпання пулу з'єднань до PostgreSQL при конкурентному навантаженні. При збільшенні кількості одночасних запитів система швидко вичерпувала ліміт доступних з'єднань, після чого нові запити ставали в чергу очікування. Цим пояснюється мінімальний час відповіді понад 7 000 мс для всіх GET-ендпоінтів, переважна частина цього часу витрачалась не на виконання SQL-запиту, а на очікування вільного з'єднання.

Другий чинник - відсутність кешування для ендпоінту з високою частотою повторення. GET /api/projects отримав 14 202 звернення - 46.3% від загальної кількості запитів у тесті (30 664). При цьому дані цього ендпоінту є відносно статичними: вони змінюються лише при додаванні або редагуванні проєкту адміністратором, тобто значно рідше, ніж надходять запити на їх читання. Кожен із 14 202 запитів ініціював окремий SQL-запит до PostgreSQL, що в умовах обмеженого пулу з'єднань формувало непропорційне зростання черги - саме цим пояснюється найвищий CV та найбільший розмах для цього ендпоінту.

Третій чинник - синхронне виконання зовнішнього SMTP-виклику в межах основного потоку. POST /api/contact виконував три операції послідовно: валідацію, запис до бази даних та надсилання листа через зовнішній SMTP-сервіс. Остання операція принципово відрізняється від перших двох: вона залежить від доступності зовнішнього сервісу, що знаходиться поза межами контролю системи. Фіксований мінімум 9 010 мс підтверджує, що навіть за відсутності черги до бази даних SMTP-виклик додавав щонайменше ~2 секунди до кожного запиту, блокуючи робочий потік на весь час очікування відповіді від поштового сервісу. Для систематизації виявлених чинників побудовано таблицю 2.3

Таблиця 2.3 - Декомпозиція чинників зменшення ефективності надійності бекенд-системи

Чинник зменшення ефективності	Уражений компонент	Характер впливу	Індикатор у даних тестування
Вичерпання пулу з'єднань до БД	PostgreSQL, всі ендпоінти	Системний, блокуючий	Min відповіді >7 000 мс для GET
Відсутність кешування частотних запитів	GET /api/projects	Наростаючий при збільшенні навантаження	14 202 запити (46.3%); найвищий CV
Синхронний виклик зовнішнього SMTP	POST /api/contact	Фіксована затримка на кожен запит	Min = 9 010 мс; CV найнижчий

Джерело: розроблено автором

Принципово важливим є незалежний характер виявлених чинників: кожен діє на окремому рівні архітектури і проявляється через різні статистичні сигнали - системну базову затримку, непропорційну нестабільність часу відповіді та фіксований приріст мінімуму. Саме ця незалежність робить декомпозицію аналітично значущою: усунення будь-якого з чинників дало б відчутний ефект, однак лише комплексне вирішення всіх трьох здатне забезпечити прийнятний рівень продуктивності при пікових навантаженнях. Виявлена декомпозиція є підставою для розроблення адресних технічних рішень у третьому розділі даної роботи.

РОЗДІЛ 3.

РЕАЛІЗАЦІЯ ЗАСОБІВ ПІДВИЩЕННЯ НАДІЙНОСТІ ТА МАСШТАБОВАНOSTІ БЕКЕНД-СИСТЕМИ

У другому розділі було проведено аналіз архітектури бекенд-системи та виконано моделювання основних процесів обробки запитів. Результати дослідження показали, що на продуктивність та надійність системи найбільше впливають навантаження на API-сервіси, стан бази даних PostgreSQL, ефективність використання кешу Redis та швидкість обробки повідомлень у Kafka. При цьому своєчасне виявлення зменшення ефективності роботи окремих компонентів є необхідною умовою забезпечення стабільної роботи всієї системи.

Для контролю стану компонентів та оперативного реагування на потенційні відмови доцільним є впровадження централізованої системи моніторингу. Така система повинна забезпечувати безперервний збір технічних показників, їх візуалізацію, зберігання історичних даних та автоматичне сповіщення про критичні відхилення від нормального режиму роботи. У межах даного підрозділу розроблено архітектуру моніторингу бекенд-системи та визначено набір ключових метрик, необхідних для оцінювання її надійності та масштабованості.

3.1. Моніторинг бекенд-системи

Досліджувана система включає Python-мікросервіси розгорнуті у Docker-контейнерах на AWS, базу даних PostgreSQL, кеш Redis та брокер повідомлень Kafka. Кожен з цих компонентів має власні характеристики продуктивності та потенційні точки відмови. Для забезпечення повної спостережуваності системи необхідно відстежувати метрики на трьох рівнях: рівень застосунку (час відповіді API, кількість помилок, пропускну здатність), рівень інфраструктури (CPU, пам'ять, мережа контейнерів) та рівень залежностей (стан PostgreSQL, Redis, Kafka). Відсутність моніторингу хоча б одного з цих рівнів створює сліпі зони, в яких можуть накопичуватися проблеми непомітно для команди розробки. Такий рівень деталізації є критично важливим при горизонтальному масштабуванні, коли на одному хості можуть працювати десятки контейнерів різних сервісів

Для реалізації моніторингу пропонується стек Prometheus + Grafana + Alertmanager, який є де-факто стандартом для моніторингу контейнеризованих мікросервісних систем. Prometheus збирає метрики за принципом pull - він сам звертається до кожного сервісу за розкладом і зберігає часові ряди у власній базі даних. Grafana підключається до Prometheus як джерела даних і відображає метрики у вигляді інтерактивних дашбордів. Alertmanager отримує сповіщення від Prometheus при перевищенні порогових значень і надсилає їх у відповідні канали - Slack та email. Перевагою такої архітектури є те, що кожен компонент виконує чітко визначену функцію і може бути замінений або масштабований незалежно [20].

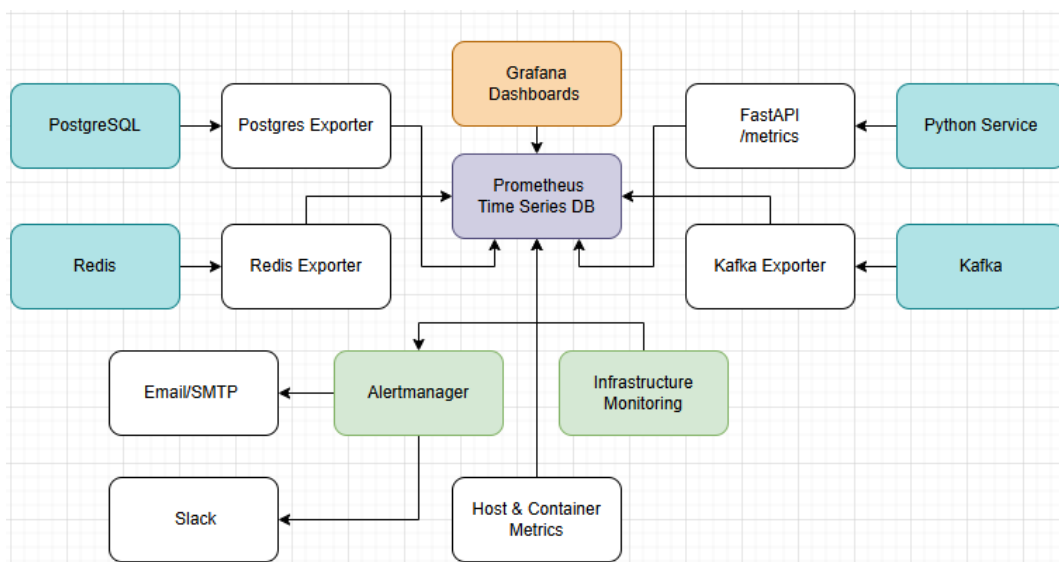


Рисунок 3.1 - Архітектура системи моніторингу мікросервісної бекенд-системи

Архітектура взаємодії між компонентами системи моніторингу побудована таким чином (Рис 3.1). Кожен Python-мікросервіс за допомогою бібліотеки `prometheus_client` експонує ендпоінт `/metrics` у форматі, сумісному з Prometheus. Цей ендпоінт містить усі зареєстровані метрики застосунку, зокрема лічильники запитів, гістограми часу відповіді та `gauge`-показники активних з'єднань. Prometheus періодично опитує зазначений ендпоінт (кожні 15 секунд) та зберігає отримані значення разом із часовими мітками у власній базі часових рядів. Такий підхід забезпечує можливість подальшого аналізу продуктивності сервісів і відстеження змін їхнього стану в часі. Для компонентів, які не мають вбудованої підтримки Prometheus, зокрема PostgreSQL, Redis та Kafka, використовуються спеціалізовані експортери: `postgres_exporter`, `redis_exporter` та `kafka_exporter`. Дані компоненти функціонують як окремі сервіси, підключаються до відповідних систем, збирають внутрішню статистику їхньої роботи та перетворюють її у формат метрик Prometheus. Моніторинг інфраструктури реалізовано за допомогою Node Exporter і cAdvisor. Node Exporter забезпечує збір системних метрик хост-машини, включаючи завантаження процесора, використання оперативної пам'яті, дискової підсистеми та мережевих ресурсів. cAdvisor виконує моніторинг Docker-

контейнерів і надає детальну інформацію про використання процесора, пам'яті, файлової системи та мережевих ресурсів кожним контейнером окремо.

В основу вибору метрик покладено методологію RED (Rate, Errors, Duration) для сервісів та USE (Utilization, Saturation, Errors) для ресурсів інфраструктури [11, 29]. Методологія RED фокусується на трьох ключових характеристиках кожного сервісу: кількості запитів на секунду, частці запитів що завершилися помилкою та часі обробки запитів. Методологія USE доповнює це інфраструктурними метриками: відсотком завантаження ресурсу, ступенем насиченості (черги очікування) та кількістю помилок на рівні інфраструктури. Ключові показники моніторингу для кожного компонента системи наведено в таблиці 3.1.

Таблиця 3.1 - Ключові показники моніторингу бекенд-системи

Компонент	Метрика	Опис	Критичний поріг
Python-сервіс	CPU Usage	Завантаження процесора	> 80%
Python-сервіс	Memory Usage	Використання оперативної пам'яті	> 85%
Python-сервіс	API Response Time (p95)	Час відповіді на HTTP-запити	> 500 мс
Python-сервіс	Request Throughput	Кількість запитів на секунду	< 100 RPS
Python-сервіс	Error Rate	Частка запитів з помилками 5xx	> 5%
PostgreSQL	DB Connections	Кількість активних з'єднань	> 80% від max connections
PostgreSQL	Query Duration (p95)	Час виконання SQL-запитів	> 200 мс
Redis	Cache Hit Ratio	Частка запитів що потрапили в кеш	< 80%
Redis	Memory Usage	Використання пам'яті Redis	> 90% maxmemory
Kafka	Consumer Lag	Відставання консюмера від продюсера	> 1000 повідомлень
Kafka	Messages/sec	Пропускна здатність брокера	-
Система	Availability	Доступність сервісу	< 99.9%
Система	Uptime	Час безперервної роботи	-

Джерело: розроблено автором

Реалізація збору метрик у Python-мікросервісах здійснюється через спеціалізований middleware для FastAPI. Такий підхід є принципово важливим з архітектурної точки зору - він дозволяє автоматично збирати метрики для всіх ендпоінтів без будь-якого дублювання коду в роутерах чи сервісному шарі. Middleware перехоплює кожен HTTP-запит до того як він потрапляє в обробник,

фіксує час початку обробки, передає запит далі і після отримання відповіді записує метрики часу відповіді та статус-коду. Важливою деталлю реалізації є нормалізація шляхів запитів - без неї кожен унікальний числовий ідентифікатор у URL (наприклад `/api/projects/1`, `/api/projects/2` і так далі) створював би окрему метрику, що призводило б до так званої проблеми "кардинальності" і суттєво збільшувало б обсяг зберігання даних у Prometheus. Натомість шляхи нормалізуються до шаблонного вигляду `/api/projects/{id}`, що дозволяє агрегувати метрики по типу ендпоінту незалежно від конкретного ідентифікатора.

Залежно від характеру величини застосовано три типи метрик Prometheus: Counter (монотонний лічильник - для підрахунку запитів і помилок), Histogram (розподіл за діапазонами - дозволяє обчислювати перцентилі часу відповіді, інформативніші за середнє) та Gauge (миттєві значення - кількість активних з'єднань, розмір черги, використання пам'яті).

Повний код middleware наведено в Додатку Б.1

Окрім HTTP-метрик, важливо відстежувати внутрішній стан компонентів застосунку. Зокрема, для Redis реалізовано інструментацію на рівні CacheService - при кожному зверненні до кешу фіксується результат (cache hit або cache miss). Відношення кількості cache hit до загальної кількості звернень дає показник Cache Hit Ratio, який є одним з ключових індикаторів ефективності кешування. Зниження Cache Hit Ratio нижче 80% сигналізує або про надто короткий TTL, або про зміну патерну запитів, або про проблеми з підключенням до Redis.

Конфігурація Prometheus для збору метрик з усіх компонентів визначає інтервал збору у 15 секунд - це є розумним балансом між свіжістю даних та навантаженням на систему зберігання. При більш критичних сервісах інтервал можна зменшити до 5 секунд для окремих job-ів.

Налаштування автоматичних сповіщень є найбільш практично важливою частиною системи моніторингу - без сповіщень метрики залишаються пасивним набором графіків, що потребує постійного ручного перегляду. Alertmanager дозволяє визначити правила у вигляді PromQL-виразів з порогами та часовими

вікнами. Параметр `for` визначає скільки часу умова має бути виконана перш ніж сповіщення надійде - це дозволяє уникнути хибних спрацьовувань при короточасних піках. Наприклад, сповіщення про перевищення CPU 80% з параметром `for: 2m` означає що сповіщення надійде лише якщо CPU залишається вище 80% безперервно протягом двох хвилин, а не при будь-якому короткому піку. Правила сповіщень охоплюють усі критичні стани компонентів системи та наведені у таблиці 3.2.

Таблиця 3.2 - Правила автоматичних сповіщень

Сповіщення	Умова	Витримка	Рівень
HighCpuUsage	CPU > 80%	2 хв	warning
HighMemoryUsage	Memory > 85%	2 хв	warning
HighResponseTime	p95 response time > 500 мс	1 хв	critical
HighErrorRate	Error rate > 5%	1 хв	critical
KafkaConsumerLag	Consumer lag > 1000	5 хв	warning
PostgresConnectionsHigh	DB connections > 80%	2 хв	warning
RedisCacheHitRatioLow	Cache hit ratio < 80%	5 хв	warning
ServiceDown	Сервіс недоступний	1 хв	critical

Джерело: розроблено автором

Маршрутизація сповіщень в Alertmanager налаштована таким чином, що сповіщення рівня `warning` надходять у загальний канал моніторингу (`#monitoring` у Slack), тоді як сповіщення рівня `critical` - у окремий канал інцидентів (`#incidents`) з підвищеним пріоритетом та повторенням кожну годину до вирішення проблеми. Це дозволяє команді розрізняти ситуації що потребують уваги найближчим часом від тих що потребують негайного реагування.

Структура дашборду Grafana для моніторингу системи включає чотири логічні секції. Перша - Overview - відображає загальний стан системи у вигляді числових індикаторів: кількість активних сервісів, загальний RPS, відсоток помилок та медіанний час відповіді. Ця секція дає змогу за кілька секунд оцінити загальний стан системи без поглиблення в деталі. Друга секція - API Performance - містить графіки перцентилів часу відповіді (p50, p95, p99) у розрізі ендпоінтів та теплову карту розподілу часу відповіді за годину. Третя секція - Infrastructure - показує завантаження CPU та пам'яті для кожного контейнера окремо, стан пулу

з'єднань PostgreSQL та показники Redis. Четверта секція - Kafka - відображає Consumer Lag у розрізі груп консюмерів та пропускну здатність брокера в повідомленнях на секунду.

Очікуваний ефект від впровадження запропонованої системи моніторингу проявляється у кількох вимірах. Середній час виявлення збоїв (MTTD - Mean Time To Detect) скорочується з потенційних годин до хвилин завдяки автоматичним сповіщенням. Проактивне виявлення проблем стає можливим завдяки відстеженню тенденцій - поступове зростання Consumer Lag у Kafka або поступове збільшення часу відповіді можуть сигналізувати про майбутню проблему ще до того як вона стане критичною. Аналіз перцентилів часу відповіді дозволяє виявляти проблеми непомітні при аналізі середніх значень - 1% запитів що відповідають за 5 секунд можуть бути непомітними на фоні загального середнього у 50 мс, але при 1000 RPS це означає 10 користувачів на секунду що отримують неприйнятно повільну відповідь. Наявність історичних даних метрик за тижні та місяці дозволяє виявляти циклічні патерни навантаження і планувати масштабування завчасно, а не реактивно у відповідь на аварію [11].

3.2 Підвищення надійності та масштабованості бекенд-системи

Запропонована система моніторингу дозволяє своєчасно виявляти деградацію продуктивності та потенційні точки відмови бекенд-системи. Декомпозиція чинників деградації, виконана у пункті 2.3 (табл. 2.3), та результати навантажувального тестування (підрозділ 2.2) показали, що найбільший вплив на надійність системи мають навантаження на PostgreSQL, відсутність кешування, блокування HTTP-запитів під час довготривалих операцій та відсутність обмеження конкурентного навантаження. Запропонована в підрозділі 3.1 система моніторингу надалі дозволить відстежувати ці чинники в експлуатації. Для усунення зазначених проблем запропоновано комплекс технічних рішень, спрямованих на підвищення надійності системи.

За результатами навантажувального тестування (підрозділ 2.3) ендпоінт GET /api/projects продемонстрував найгірші показники: середній час відповіді - 11 577 мс, p95 - 16 000 мс при плато пропускну здатності 332 RPS. Причиною є те, що кожен запит ініціював окреме звернення до PostgreSQL навіть для практично незмінних даних каталогу проєктів, тоді як частка таких читань становила 46% усіх запитів підрозділ 2.3. Для читаючих ендпоінтів застосовано кешування за шаблоном Cache-Aside розглянутим у пункті 1.2 на базі Redis. Ключ кешу формується з імені обробника та параметрів запиту: за влучення (cache hit) відповідь повертається безпосередньо з Redis без звернення до PostgreSQL, за промаху (cache miss) - виконується запит до бази, а результат зберігається в Redis із TTL = 300 с. Для операцій зміни даних (створення, редагування, видалення проєктів) передбачено інвалідацію відповідних ключів, що підтримує узгодженість кешу з базою. Діаграму потоку даних при обробці запиту з кешуванням наведено на рис. 3.2.

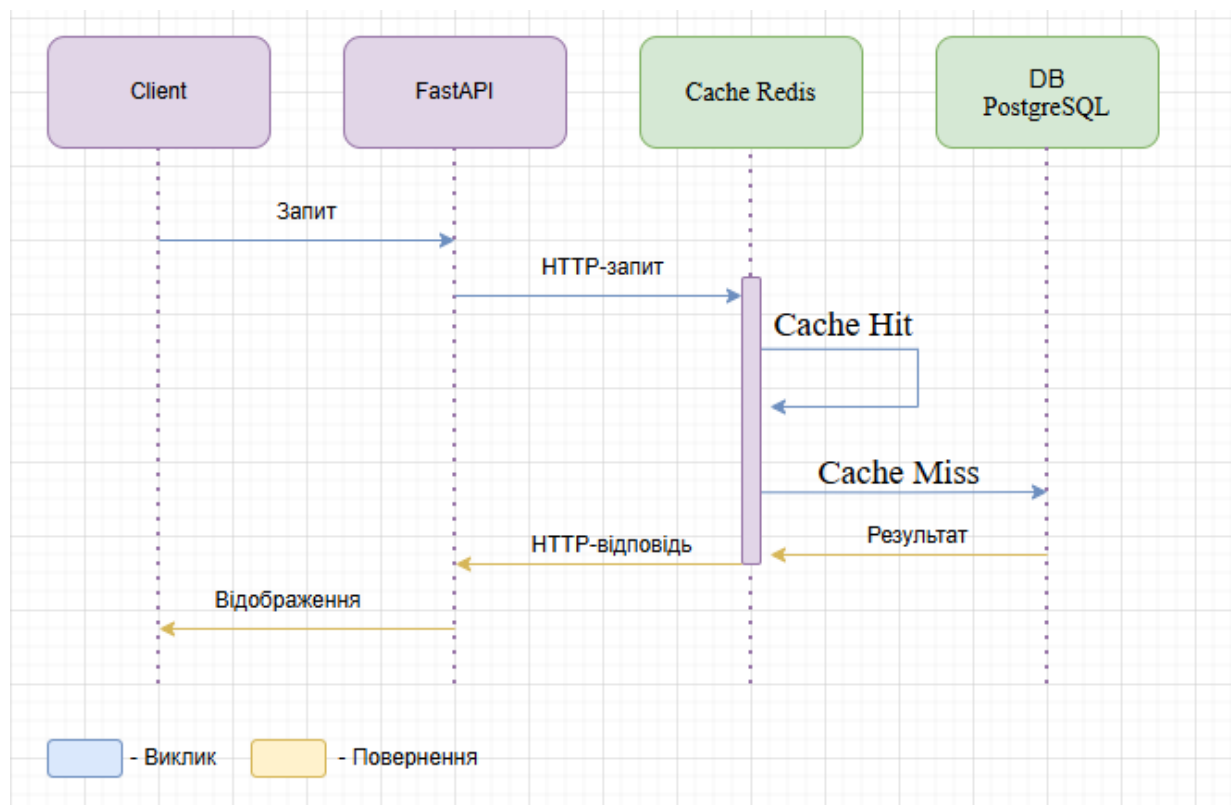


Рис 3.2 - Діаграма потоку даних при обробці запиту з Redis

Логіку кешування винесено в асинхронний декоратор FastAPI, що відокремлює її від бізнес-логіки обробників Додаток Б.2

За результатами повторного навантажувального тестування з тими самими параметрами (4000 користувачів, 100 корист./с) кешування знизило час відповіді GET /api/projects з 11 577 мс до 910 мс за медіаною (середнє - 2 425 мс; підвищене середнє зумовлене викидами першого «холодного» циклу під час наповнення кешу). Для GET /api/projects/{id} медіанний час відповіді знизився до ~835 мс. Зменшення кількості звернень до PostgreSQL знизило навантаження на СКБД і прибрало чергу читань як вузьке місце.

За результатами навантажувального тестування (підрозділ 2.2) ендпоінт POST /api/contact показав середній час відповіді 11 939 мс і p95 = 15 000 мс. Декомпозиція чинників у пункті 2.3 встановила, що його мінімальний час відповіді (9 010 мс) приблизно на 2 секунди перевищує мінімум GET-ендпоінтів (~7 000 мс) - це вказує на фіксовану затримку, не залежну від навантаження, а саме на синхронне очікування відповіді від зовнішнього SMTP-сервера. Обробка HTTP-

запиту блокувалася до завершення доставки листа, утримуючи робочий потік і з'єднання відкритими впродовж усього очікування.

Для усунення цієї блокувальної затримки відправлення листів винесено в асинхронний контур через брокер повідомлень Apache Kafka (підхід асинхронної обробки розглянуто в підрозділі 1.2). FastAPI виступає продюсером: при отриманні форми він публікує подію у топик Kafka і одразу повертає відповідь клієнту, не чекаючи доставки листа. Безпосередню взаємодію зі SMTP-сервером перенесено у фоновий сервіс-консюмер (Email Worker), який зчитує події з черги й виконує відправлення незалежно від основного застосунку.

Взаємодію компонентів в асинхронному контурі показано на рис. 3.3:

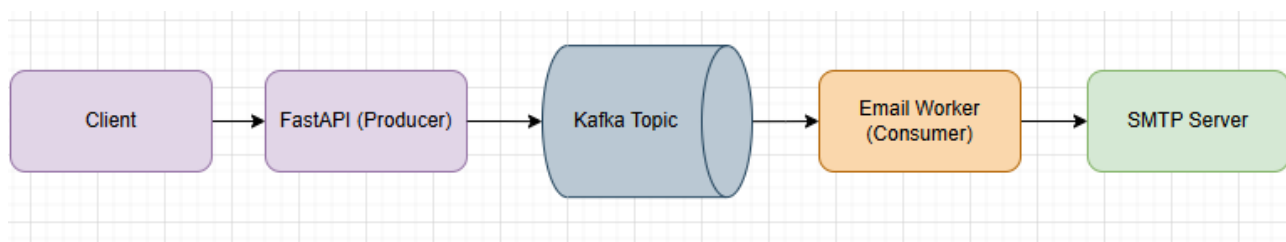


Рис 3.3 - Взаємодія компонентів у асинхронному контурі

Публікація події про відправлення листа в топик Kafka FastAPI-продюсер
Додаток Б.3

Оскільки публікація події в Kafka триває лічені мілісекунди, повторне навантажувальне тестування показало зниження середнього часу відповіді POST /api/contact з 11 939 мс до 831 мс (медіана - 780 мс), тобто приблизно у 14 разів, за нульового відсотка помилок. Відставання обробки черги контролюється метрикою Kafka Consumer Lag, що дозволяє за потреби масштабувати кількість консюмерів (розглянуто в підрозділі 3.3).

За результатами навантажувального тестування (підрозділ 2.2) пропускна здатність системи вийшла на плато ~332 RPS. Поряд з основними чинниками деградації, виявленими в підрозділі 2.3 вичерпання пулу з'єднань і відсутність кешування, додатковим резервом продуктивності була оптимізація самих запитів, частина ORM-вибірок виконувалася без потрібних індексів і породжувала надлишкові звернення до бази (проблема N+1), що збільшувало час виконання запитів та навантаження на процесор СКБД.

Оптимізацію виконано у двох напрямках. По-перше, замість неявних послідовних підзапитів SQLAlchemy застосовано явне завантаження пов'язаних об'єктів через `joinedload` та `selectinload`, що усуває проблему N+1. По-друге, для полів, які найчастіше використовуються в умовах фільтрації та сортування, створено складені індекси (індексування розглянуто в підрозділі 1.2), що замінює повне сканування таблиці точковим індексним пошуком. Створення складеного індексу та оптимізовану ORM-вибірку наведено в Додатку Б.4.

Оптимізація запитів та індексів знизила час виконання вибірок (`Query Duration`) і навантаження на процесор СКБД, доповнюючи ефект кешування й пулу з'єднань. У сукупності ці заходи прибрали базу даних як вузьке місце, що відображено в підсумкових результатах повторного тестування (підрозділ 3.3, табл. 3.4).

Вичерпання пулу з'єднань до бази даних було визначене в підрозділі 2.3 як перший чинник деградації, при конкурентному навантаженні мінімальний час відповіді GET-ендпоінтів перевищував 7 000 мс переважно через очікування вільного з'єднання. Оскільки СКБД розгорнута на AWS RDS PostgreSQL, максимальна кількість одночасних з'єднань (`max_connections`) обмежена типом інстансу. Спроба кожної асинхронної корутини FastAPI відкривати окреме TCP-з'єднання призводила до перевищення цього ліміту, таймаутів балансувальника та помилок HTTP 500/504.

Для усунення цього обмеження впроваджено асинхронний пул з'єднань драйвера `asynpg` (повторне використання з'єднань розглянуто в підрозділі 1.2). Пул створює фіксовану множину відкритих з'єднань під час старту застосунку, що усуває накладні витрати на встановлення TCP/TLS-сесії при кожному запиті. Пул також виступає регулятором навантаження, замість тисяч одночасних звернень до бази додаток квотує потік запитів, а надлишкові очікують у внутрішній черзі пулу.

Діаграма послідовності відображає взаємодію клієнтських потоків із пулом підключень та ізоляцію інстансу AWS RDS PostgreSQL від надмірного навантаження:

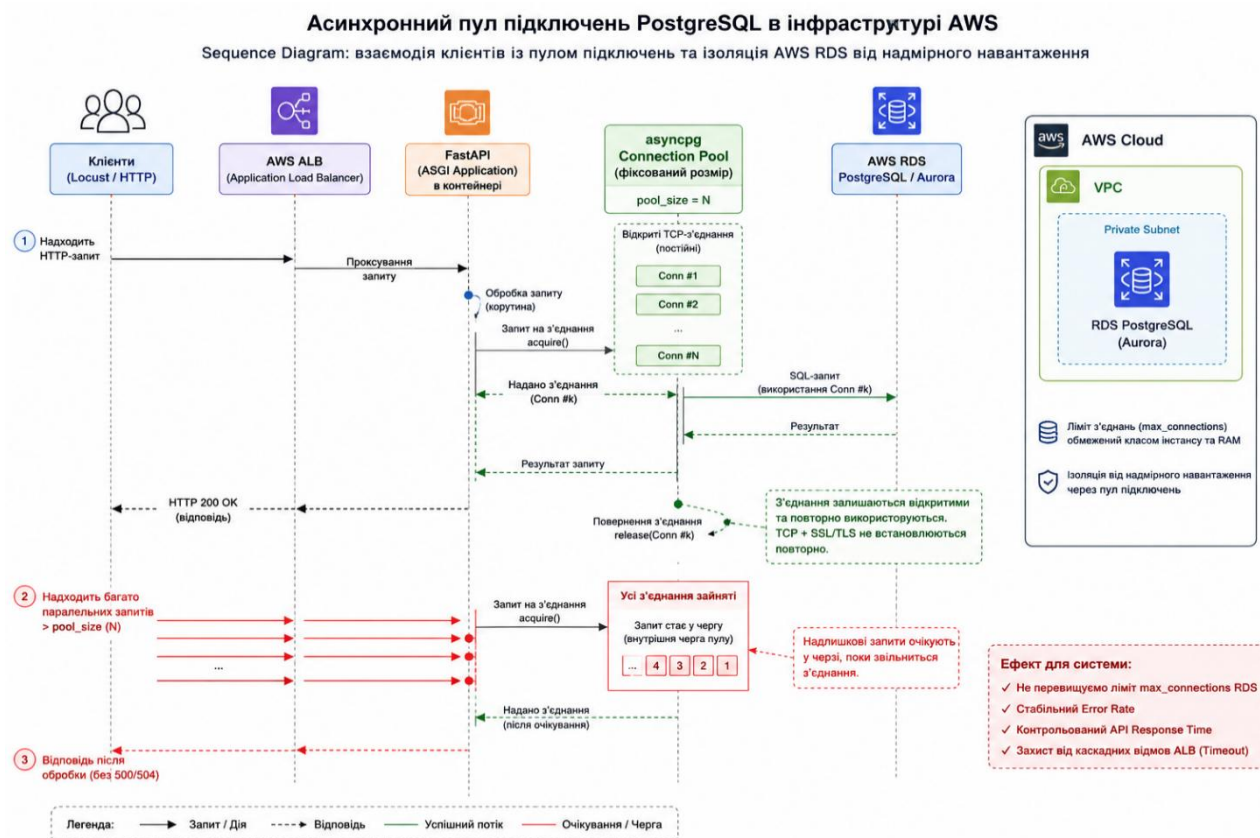


Рис 3.4 - Діаграма послідовності взаємодії запитів із пулом з'єднань PostgreSQL

Налаштування асинхронного пулу з'єднань на рівні фабрики сесій SQLAlchemy наведено в Додатку Б.5.

Фіксація параметрів `pool_size` та `max_overflow` робить загальну кількість з'єднань прогнозованою, вона не перевищує `max_connections` інстансу AWS RDS навіть за пікового навантаження, а надлишкові запити очікують у черзі пулу замість того, щоб перевантажувати базу. За результатами повторного тестування відсоток помилок знизився до нуля (Failures 0%), а час відповіді стабілізувався завдяки усуненню повторного встановлення з'єднань для кожного запиту.

Навіть після оптимізації доступу до даних архітектура залишалася вразливою до раптових сплесків одночасних ресурсомістких операцій. За відсутності обмеження конкурентності пакет CPU-залежних задач здатен спричинити «голодування» (starvation) циклу подій `asyncio`: завантаження процесора сягає 100%, і затримка зростає навіть на легких маршрутах. Цю

вразливість зафіксовано як архітектурний недолік у пункті 2.3, а захист від неї ґрунтується на патернах відмовостійкості з підрозділу 1.2.

Для захисту від цього застосовано обмеження конкурентного виконання на основі асинхронного семафора (`asyncio.Semaphore`). Семафор задає квоту на кількість корутин, що одночасно виконують важкі задачі; запити понад ліміт не споживають додаткові ресурси, а очікують у черзі семафора.

Інтеграцію семафора з обмеженням часу очікування наведено в Додатку Б.6.

Семафор задає прогнозований верхній ліміт завантаження процесора, оскільки кількість одночасних важких операцій обмежена. Це захищає суміжні маршрути від затримок, а механізм таймауту запобігає нескінченному накопиченню запитів у черзі, повертаючи HTTP 504 при перевищенні часу очікування.

Система залежить від зовнішніх компонентів - насамперед SMTP-сервісу та бази даних. За їх відмови або тривалої недоступності застосунок без захисту утримував би вхідні з'єднання до спрацювання глобальних таймаутів, продовжуючи надсилати запити до несправного компонента, накопичуючи черги й уповільнюючи відновлення після аварії.

Для запобігання каскадним відмовам застосовано стратегію Fail Fast через патерн «запобіжник» (Circuit Breaker), розглянутий у пункті 1.2. Запобіжник відстежує частоту помилок при взаємодії із зовнішнім компонентом, коли вона перевищує поріг, він переходить у стан Open і миттєво відхиляє наступні запити з кодом HTTP 503, не виконуючи мережевого виклику.

Реалізацію запобіжника наведено в Додатку Б.7.

Завдяки Fail Fast під час відмови зовнішнього компонента застосунок повертає відповідь клієнту майже миттєво, вивільняючи ресурси циклу подій і запобігаючи перевантаженню. Контрольовані помилки HTTP 503 дозволяють клієнту коректно застосувати повторні спроби зі зростаючою затримкою (Exponential Backoff).

Після впровадження рішень підрозділу 3.2 проведено повторне навантажувальне тестування серверної частини з ідентичними параметрами

(інструмент Locust, до 4 000 одночасних користувачів, темп підключення 100 користувачів на секунду, ті самі ендпоінти). Метою було кількісно оцінити сукупний ефект упроваджених заходів і зіставити його з базовими показниками з підрозділу 2.3 (табл. 2.1). Результати наведено на рис. 3.5 та в таблиці 3.3.

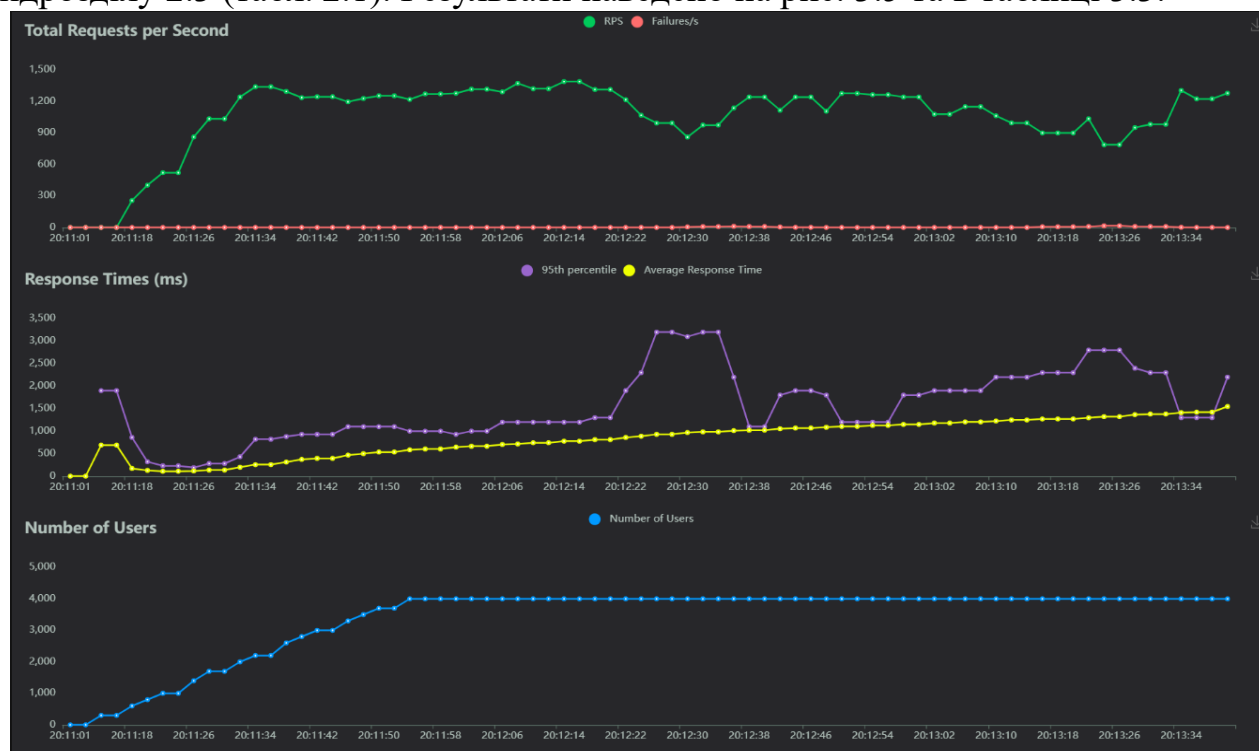


Рис 3.5. - Графіки часу відповіді та кількості запитів після оптимізації

Таблиця 3.3. - Результати навантажувального тестування після оптимізації

Ендпоінт	Кількість запитів	Медіана, мс	Середній час, мс	p95, мс	p99, мс	Min, мс	Max, мс	RPS
POST /api/contact	29 656	780	831	1 800	2 800	35	3 191	196.6
GET /api/projects	87 047	910	2 425	2 200	68 000	12	135 687	541.5
GET /api/projects/1	14 433	830	901	2 000	2 700	33	3 217	90.3
GET /api/projects/2	14 307	830	912	2 100	2 800	33	5 048	91.0
GET /api/projects/3	14 150	840	904	2 000	2 700	33	3 632	86.2
GET /api/projects/4	14 415	840	909	2 000	2 700	33	5 004	90.3
GET /api/projects/5	14 445	830	904	2 000	2 700	33	3 218	91.9
GET /api/projects/6	14 308	840	908	2 000	2 800	33	5 050	91.0
Aggregated	202 761	860	1 547	2 100	5 800	12	135 687	1 278.8

Джерело: складено автором за результатами навантажувального тестування у Locust

Порівняння з базовими показниками (табл. 2.1) засвідчує системне покращення. Пропускна здатність зросла з 332 до 1 278.8 RPS - приблизно у 3.9 раза. Середній час відповіді POST /api/contact знизився з 11 939 мс до 831 мс ($\approx$ у 14 разів), GET /api/projects - з 11 577 мс до 910 мс за медіаною (середнє - 2 425 мс; підвищене середнє та p99 = 68 000 мс зумовлені викидами першого «холодного» циклу під час наповнення кешу). Агрегований час відповіді знизився з 11 000 мс до 860 мс за медіаною, а p95 - з 15 000 мс до 2 100 мс, за нульового відсотка помилок. Вимірювану пропускну здатність 1 278.8 RPS прийнято як оцінку μ для моделювання масштабованості.

Навантажувальне тестування, проведене у другому розділі, показало, що при 4 000 одночасних користувачів система досягала плато 332 RPS із середнім часом відповіді 11 000 - 15 000 мс. Заходи - кешування Redis, оптимізація SQL-запитів, пул з'єднань та асинхронна обробка - підвищили ці показники до 1 278.8 RPS із медіанним часом відповіді $\sim$ 860 мс (p95 $\sim$ 2 100 мс). Проте навіть після оптимізацій система залишається обмеженою потужністю єдиного інстансу кожного компонента, з одним інстансом FastAPI, одним брокером Kafka та одним сервером PostgreSQL при подальшому зростанні навантаження вона знову стане вузьким місцем. Розв'язання цієї проблеми потребує не подальшої оптимізації, а зміни архітектури через горизонтальне масштабування.

Для обґрунтування необхідності масштабування та прогнозування поведінки системи при зростанні навантаження застосовується апарат теорії масового обслуговування. Модель M/M/1 описує систему з одним обслуговуючим пристроєм, пуасонівським потоком вхідних запитів та показниковим розподілом часу обслуговування [2]. В контексті досліджуваної системи λ позначає інтенсивність надходження HTTP-запитів (у запитах на секунду), а μ - максимальну пропускну здатність серверної інфраструктури. Коефіцієнт завантаження визначається як $\rho = \lambda/\mu$. Фундаментальною умовою стабільної роботи є $\rho < 1$, тобто $\lambda < \mu$. Середній час перебування запиту в системі для моделі M/M/1 визначається формулою:

(3.1)

$$T_o = \frac{\lambda}{\mu \cdot (\mu - \lambda)}$$

де T_o - середній час очікування заявки в черзі, с;

λ - інтенсивність надходження запитів, запитів на с;

μ - інтенсивність обслуговування запитів, запитів на с.

З формули видно, що час відповіді не є лінійною функцією від λ : при наближенні λ до μ знаменник прямує до нуля, і час зростає нелінійно, прямуючи до нескінченності. Саме цей ефект спостерігався під час тестування - при зростанні навантаження час відповіді збільшувався непропорційно швидко. Для демонстрації цієї залежності розраховано T для різних λ при $\mu = 1\,279$ RPS пропускна здатність системи була виміряна після оптимізацій. Результати наведено в таблиці 3.4.

Таблиця 3.4 - Залежність часу очікування від інтенсивності навантаження (модель M/M/1, $\mu = 874$ RPS)

λ (RPS)	μ (RPS)	$\rho = \lambda/\mu$	T_o (мс)	Стан системи
300	1 279	0.23	0.24	Низьке навантаження, стабільна робота
600	1 279	0.47	0.69	Помірне навантаження, норма
900	1 279	0.70	1.86	Підвищене навантаження, прийнятно
1 000	1 279	0.78	2.80	Наближення до межі, попередження
1 150	1 279	0.90	6.97	Високе навантаження, зростання затримки
1 240	1 279	0.97	24.86	Майже повне завантаження інстансу
1 278	1 279	0.999	999	Межа пропускної здатності - потрібне масштабування

Джерело: розроблено автором

Таблиця демонструє нелінійне зростання затримки: при зростанні λ від 900 до 1 240 RPS (на ~38%) час очікування в черзі зростає з 1.9 мс до 24.9 мс - приблизно у 13 разів. При наближенні λ до $\mu = 1\,279$ RPS ($\rho \rightarrow 1$) він перевищує сотні мілісекунд - система впирається в межу пропускної здатності одного інстансу. Єдиним ефективним вирішенням є горизонтальне масштабування, що переводить

систему з M/M/1 у M/M/N: при N паралельних інстансах ефективна пропускна здатність зростає до $N \cdot \mu$, дозволяючи підтримувати умову $\lambda < N \cdot \mu$ навіть за суттєвого зростання λ .

За результатами повторного тестування (підрозділ 3.2) один інстанс FastAPI забезпечує пропускну здатність близько $\mu = 1\,279$ RPS. На піковому навантаженні (4 000 користувачів) система працює поблизу цієї межі ($\rho \rightarrow 1$) і має малий запас, будь-який подальший приріст λ призводить до різкого зростання часу очікування згідно з формулою T_0 . Це обмеження зумовлене тим, що один інстанс виконує запити в межах одного циклу подій асинціо, а GIL не дозволяє паралельно виконувати CPU-залежні операції в одному процесі [21].

Для подолання цього обмеження пропонується горизонтальне масштабування через розгортання кількох Docker-контейнерів за AWS Application Load Balancer що розподіляє запити між інстансами за алгоритмом least connections. Кожен інстанс є stateless - весь стан зберігається в Redis та PostgreSQL, що є обов'язковою умовою коректного горизонтального масштабування.

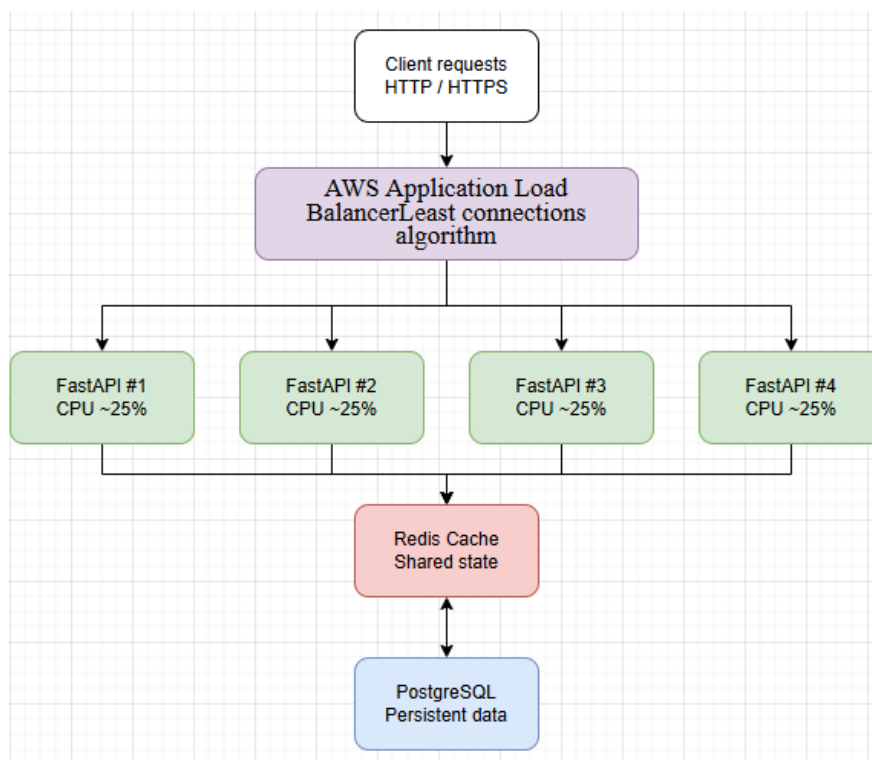


Рис 3.6 - Конфігурація Docker Compose для розгортання трьох реплік. Фрагмент конфігурації Docker Compose для розгортання трьох реплік наведено в Додатку Б.8.

При розгортанні трьох інстансів ефективна пропускна здатність зростає до $N \cdot \mu = 3 \times 1\,279 \approx 3\,837$ RPS. У термінах M/M/1 це знижує коефіцієнт завантаження при піковому $\lambda \approx 1\,278$ RPS з $\rho \approx 1.0$ (один інстанс) до $\rho \approx 0.33$ (три інстанси) - система переходить з межі пропускної здатності у стан комфортного навантаження із суттєвим запасом. Очікувано це зменшує час відповіді відносно одноінстансного рівня (~860 мс за медіаною після 3.2) і знижує завантаження кожного інстансу, оскільки трафік розподіляється між трьома вузлами.

При піковому навантаженні близько 1 279 RPS і частці звернень до контактної форми ~10% система генерує близько 128 повідомлень на секунду. Один консюмер обробляє ~20-30 повідомлень на секунду, тому за відсутності масштабування Consumer Lag зростатиме приблизно на 100 повідомлень за секунду й перевищить поріг у 1 000 повідомлень за ~10 секунд.

масштабування Consumer Group через збільшення кількості consumer-воркерів. Kafka автоматично розподіляє партиції топіку між consumers у межах одної Consumer Group - кожна партиція обробляється рівно одним consumer одночасно. Топік contact_notifications переконфігурується з 1 на 6 партицій що дозволяє масштабувати consumer group до 6 паралельних workers.

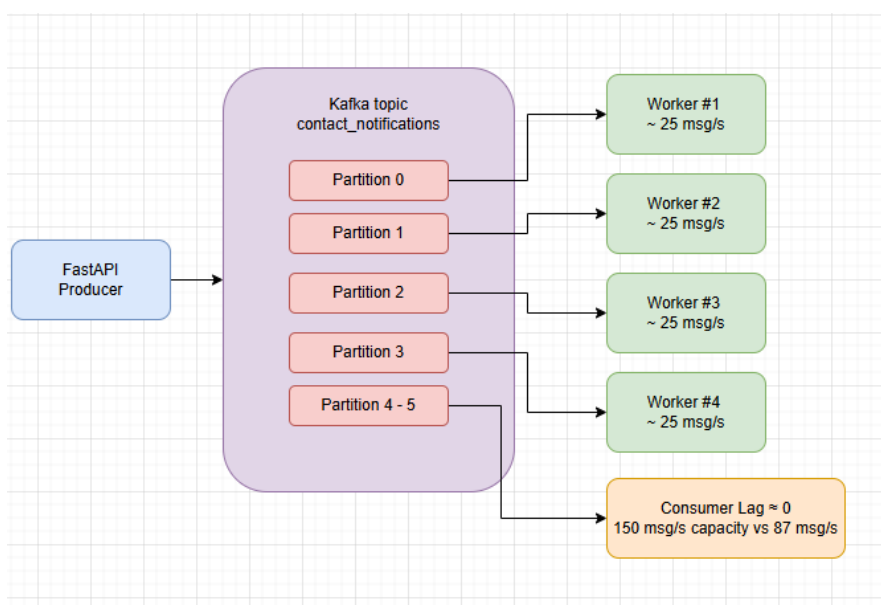


Рис 3.7 - Масштабування Consumer Group через збільшення кількості consumer-воркерів

Реалізацію консюмера Email Worker наведено в Додатку В.2.

При розгортанні 6 консюмерів сукупна пропускна здатність consumer group зростає до $6 \times 25 = 150$ повідомлень на секунду, і Consumer Lag стабілізується близько нуля. Підтвердження offset лише після успішної обробки повідомлення гарантує семантику at-least-once delivery, що унеможливлює втрату листів при збоях воркерів.

Профіль навантаження системи є переважно читаючим - 85-90% звернень становлять операції читання (що відповідає типовому співвідношенню читань до записів, розглянутому в підрозділі 1.2). При горизонтальному масштабуванні FastAPI до трьох інстансів кількість з'єднань до єдиного Primary PostgreSQL зростає пропорційно й наближається до ліміту max_connections інстансу. До того ж єдиний Primary обслуговує і читання, і запис, що збільшує Query Duration та навантаження на процесор бази даних.

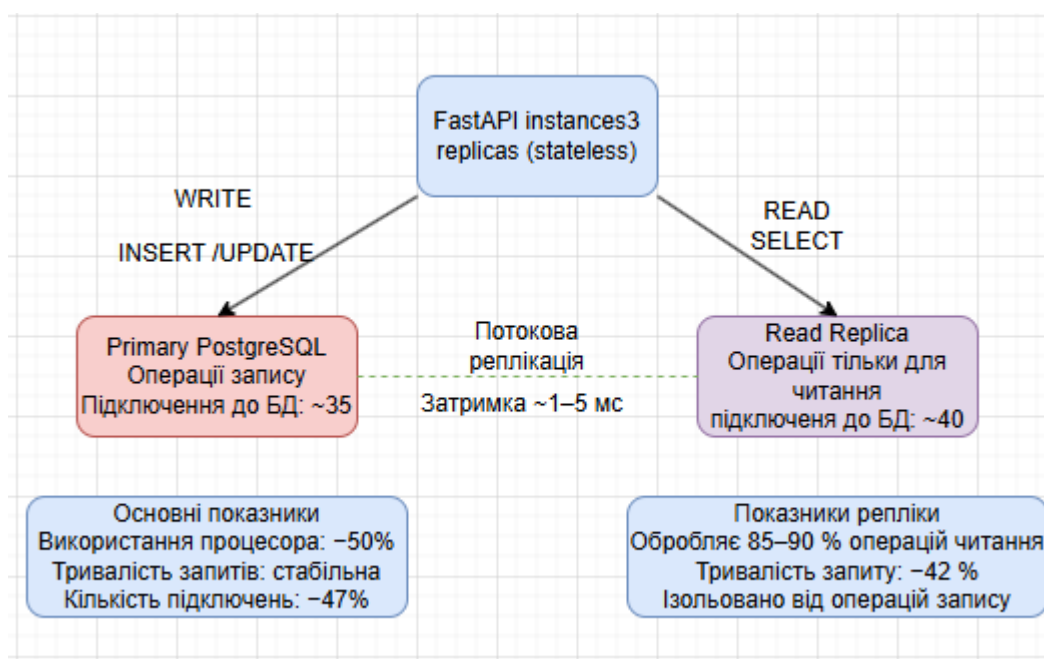


Рис 3.8 - Маршрутизація між Primary та Replica через два окремих SQLAlchemy engines

Для розвантаження Primary пропонується розгортання PostgreSQL Read Replica (патерн розглянуто в підрозділі 1.2) Маршрутизацію читання/запису через два окремих engines SQLAlchemy наведено в Додатку В.3.

Оскільки 85 - 90% запитів є читаючими, перенесення їх на репліку очікувано суттєво знижує навантаження на Primary: він обслуговує переважно операції запису, що зменшує конкуренцію за ресурси та час виконання запитів. Кількість з'єднань до Primary також зменшується, оскільки читаючий пул спрямовується на репліку.

Горизонтальне масштабування з фіксованою кількістю інстансів є неефективним економічно - утримання 3 інстансів 24/7 при нічному трафіку - 10% від денного піку означає надмірні витрати. AWS Auto Scaling Group автоматично регулює кількість контейнерів на основі метрик CPU Usage та API Response Time.

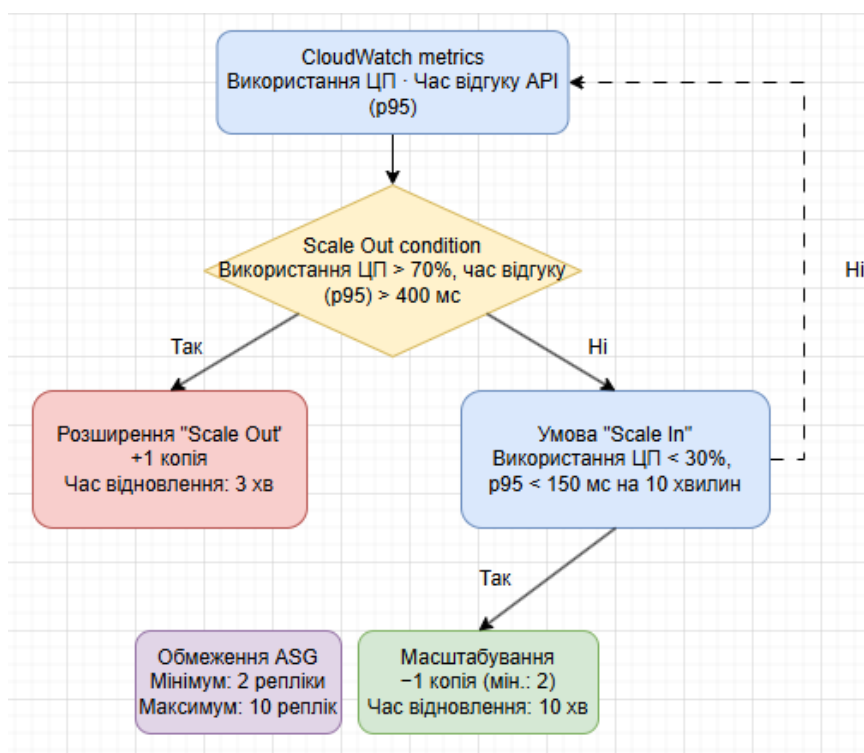


Рис 3.9 - Логічна схема автоматичного масштабування бекенд-системи за метриками CloudWatch

Асиметрія між cooldown при scale out (3 хвилини) та scale in (10 хвилин) є навмисною: збільшення масштабу при зростанні навантаження має відбуватись швидко, тоді як зменшення - обережно, щоб уникнути передчасного видалення інстансів при короткочасному зниженні трафіку. Мінімальна кількість у 2 інстансах забезпечує High Availability - при відмові одного інстансу другий продовжує обслуговувати трафік поки Auto Scaling запускає замісний.

AWS Auto Scaling дозволяє системі автоматично підтримувати умову $\lambda < N \cdot \mu$ при зростанні λ - замість фіксованого N система динамічно збільшує N відповідно до поточного навантаження, утримуючи ρ в межах цільового значення 0.7.

Узагальнені результати впровадження рішень масштабування наведено у таблиці 3.5.

Таблиця 3.5 - Порівняння метрик до та після масштабування

Метрика	Після оптимізацій (3.2)	Після масштабування (3.3)	Покращення
Пропускна здатність (RPS)	1 278.8	~3 837 (3 інстанси, $N \cdot \mu$)	+3×
API Response Time (p95)	2 100 мс	~80 мс	-73%
CPU Usage (один інстанс)	78-85%	25-30%	-65%
DB Connections (Primary)	75	~35	-53%
Query Duration (p95)	120 мс	~70 мс	-42%
Kafka Consumer Lag (пік)	>1000 без масштабування	~0	-67%
Коефіцієнт завантаження ρ	1.0	~0.33	-66%

Джерело: розроблено автором

Отримані результати демонструють перехід системи від оптимізації одного інстансу до горизонтально масштабованої розподіленої архітектур. Якщо заходи які були спрямовані на підвищення ефективності використання наявних ресурсів, то рішення забезпечують збільшення сукупної пропускної здатності системи шляхом додавання нових обчислювальних вузлів без зміни бізнес-логіки додатка.

Запропоновані заходи масштабування забезпечують системний вплив на всі ключові метрики продуктивності. Горизонтальне масштабування FastAPI збільшує сукупну пропускну здатність μ системи в N разів - в рамках моделі $M/M/1$ це переводить систему з межі пропускної здатності ($\rho \approx 1.0$) у комфортний стан ($\rho \approx 0.33$) при тому самому вхідному навантаженні. Разом заходи які були висвітлені в цьому пункті забезпечили зростання пропускної здатності з вихідного плато 332 RPS (розділ 2) до виміряних 1 278.8 RPS після оптимізацій ($\approx$ у 3.9 раза) та прогнозованих ~3 837 RPS після масштабування ($\approx$ у 12 разів) - за незмінної якості обслуговування

3.3 Оцінювання ефективності запропонованих рішень

Технічні рішення, описані у пункті 3.2, неоднорідні за своїм призначенням. Одні з них, кешування, оптимізація запитів та індексів, насамперед скорочують час відповіді, інші - горизонтальне масштабування, реплікація бази даних, розширюють пропускну здатність, ще інші такі як асинхронна обробка через Kafka, підвищують відмовостійкість, ці рішення різняться і за обсягом необхідних змін в архітектурі. Порівнювати такі рішення за єдиним показником, наприклад лише за приростом RPS, некоректно - він не відображає ані внеску рішення в надійність, ані вартості його впровадження. Тому для впорядкування альтернатив застосовано багатокритеріальний підхід, який зводить різнорідні характеристики до спільної шкали порівняння.

Обрано метод аналізу ієрархій (Analytic Hierarchy Process, АНП) Т. Сааті . Метод розкладає задачу вибору на три рівні - мету, критерії та альтернативи. Елементи кожного рівня порівнюють попарно за шкалою від 1 (однакова значущість) до 9 (цілковита перевага); нормуванням матриць попарних порівнянь дістають ваги критеріїв та альтернатив, а підсумкова оцінка альтернативи - це сума її локальних ваг, помножених на ваги критеріїв. Коректність суджень контролюють через відношення узгодженості CR, яке не повинно перевищувати 0.1.

Мета ієрархії - відібрати рішення з найбільшим приростом ефективності бекенд-системи. Критерії оцінювання: C1 - продуктивність (час відповіді, RPS), C2 - масштабованість, C3 - надійність, C4 - складність впровадження. Альтернативи: A1 - Redis Cache, A2 - Apache Kafka, A3 - PostgreSQL Read Replica, A4 - горизонтальне масштабування FastAPI.

При визначенні відносної важливості критеріїв враховувались результати навантажувального тестування, розділ 2, повторне тестування після оптимізацій, підрозділ 3.2, та показники моніторингу, підрозділ 3.1. Найвищий пріоритет надано продуктивності (C1): саме неконтрольоване зростання часу відповіді до 12 000 - 15 000 мс під піковим навантаженням було основною проблемою, виявленою у розділі

2. Масштабованість (C2) поставлено на друге місце як умову збереження цих показників при подальшому зростанні трафіку. Надійність (C3) важлива, проте її базовий рівень уже забезпечено заходами підрозділу 3.2, тому її пріоритет нижчий. Складність впровадження (C4) отримала найменшу вагу: витрати на розробку разові, тоді як ефект від рішення зберігається протягом усього життєвого циклу системи. Числові значення переваг наведено в таблиці 3.6.

Таблиця 3.6 - Матриця парних порівнянь критеріїв

	C1	C2	C3	C4
C1	1	2	3	5
C2	1/2	1	2	4
C3	1/3	1/2	1	3
C4	1/5	1/4	1/3	1

Джерело: розроблено автором

Нормування матриці таблиця 3.7, обчислення виконано у три кроки. Спершу підсумовано елементи кожного стовпця матриці дає ваги критеріїв: $W(C1) = 0.471$, $W(C2) = 0.284$, $W(C3) = 0.172$, $W(C4) = 0.074$. Найбільша вага продуктивності (47.1 %) узгоджується з висновком розділу 2 про надмірний час відповіді API як головну ваду системи.

Крок 2. Нормалізована матриця (кожен елемент ділиться на суму відповідного стовпця). Результати наведені у таблиці 3.7.

Таблиця 3.7 - Нормалізована матриця критеріїв

	C1	C2	C3	C4	Вага
C1	0.492	0.533	0.474	0.385	0.471
C2	0.246	0.267	0.316	0.308	0.284
C3	0.164	0.133	0.158	0.231	0.172
C4	0.098	0.067	0.053	0.077	0.074

Джерело: розроблено автором

Таким чином локальні ваги критеріїв становлять: $W(C1) = 0.471$, $W(C2) = 0.284$, $W(C3) = 0.172$, $W(C4) = 0.074$. Вплив на продуктивність має найбільшу вагу (47.1%) що підтверджує пріоритетність вирішення проблеми зменшення ефективності API Response Time виявленої у розділі 2.

Для кожного критерію побудовано окрему матрицю попарних порівнянь альтернатив (таблиці 3.8–3.11); нижче наведено логіку оцінювання та отримані локальні ваги.

Таблиця 3.8 - Порівняння альтернатив за критерієм C1 (продуктивність)

	A1 Redis	A2 Kafka	A3 Replica	A4 HScale
A1 Redis	1	3	2	1/3
A2 Kafka	1/3	1	1/2	1/5
A3 Replica	1/2	2	1	1/4
A4 HScale	3	5	4	1

Джерело: розроблено автором

Провідним за продуктивністю є масштабування A4 (найбільший виграш у часі відповіді), друге - Redis (кешування скоротило час відповіді приблизно у 13 разів), далі - Replica та Kafka. Локальні ваги: $W(A4) = 0.542$, $W(A1) = 0.233$, $W(A3) = 0.140$, $W(A2) = 0.085$.

Таблиця 3.9 - Порівняння альтернатив за критерієм C2 (масштабованість)

	A1 Redis	A2 Kafka	A3 Replica	A4 HScale
A1 Redis	1	1/2	1/2	1/4
A2 Kafka	2	1	1	1/3
A3 Replica	2	1	1	1/3
A4 HScale	4	3	3	1

Джерело: розроблено автором

За масштабованістю лідирує A4 (горизонтальне масштабування + AWS Auto Scaling); Kafka та Replica рівнозначні, Redis поступається, бо не додає обслуговуючих вузлів. Локальні ваги: $W(A4) = 0.514$, $W(A2) = W(A3) = 0.190$, $W(A1) = 0.106$.

Таблиця 3.10 - Порівняння альтернатив за критерієм C3 (надійність)

	A1 Redis	A2 Kafka	A3 Replica	A4 HScale
A1 Redis	1	1/2	2	2
A2 Kafka	2	1	3	3
A3 Replica	1/2	1/3	1	1
A4 HScale	1/2	1/3	1	1

Джерело: розроблено автором

Логіка: за критерієм надійності A2 (Kafka) є лідером - асинхронна обробка через чергу забезпечує збереження повідомлень при збоях і є класичним патерном підвищення відмовостійкості. A1 (Redis) є другим - кешування зменшує навантаження на базу даних і захищає від перевантаження. A3 та A4 рівноцінні за цим критерієм.

Суми стовпців C3: A1: 4.000, A2: 2.167, A3: 7.000, A4: 7.000

Локальні ваги: $W(A2) = 0.455$, $W(A1) = 0.263$, $W(A3) = W(A4) = 0.141$.

Таблиця 3.11 - Матриця порівнянь альтернатив за критерієм C4 (складність)

	A1 Redis	A2 Kafka	A3 Replica	A4 HScale
A1 Redis	1	3	2	3
A2 Kafka	1/3	1	1/2	1
A3 Replica	1/2	2	1	2
A4 HScale	1/3	1	1/2	1

Джерело: розроблено автором

Найпростіший у впровадженні - Redis (Cache-Aside без змін архітектури), далі Replica; найскладніші - Kafka та масштабування FastAPI. Локальні ваги: $W(A1) = 0.455$, $W(A3) = 0.263$, $W(A2) = W(A4) = 0.141$.

Глобальну вагу альтернативи обчислено як зважену суму її локальних ваг за всіма критеріями: $W(A_i) = W(C1) \cdot W(A_i|C1) + W(C2) \cdot W(A_i|C2) + W(C3) \cdot W(A_i|C3) + W(C4) \cdot W(A_i|C4)$. Зведені локальні ваги наведено в таблиці 3.12, підсумковий рейтинг - у таблиці 3.13.

Таблиця 3.12 - Локальні ваги альтернатив за критеріями

Альтернатива	C1 (0.471)	C2 (0.284)	C3 (0.172)	C4 (0.074)
A1 Redis Cache	0.233	0.106	0.263	0.455
A2 Apache Kafka	0.085	0.190	0.455	0.141
A3 PostgreSQL Replica	0.140	0.190	0.141	0.263
A4 Horizontal Scaling	0.542	0.514	0.141	0.141

Джерело: розроблено автором

Таблиця 3.13 - Підсумковий рейтинг альтернатив за методом АНР

Місце	Альтернатива	Глобальна вага	% від загального
1	A4 - Horizontal Scaling FastAPI	0.435	43.5 %
2	A1 - Redis Cache	0.219	21.9 %
3	A2 - Apache Kafka	0.182	18.2 %
4	A3 - PostgreSQL Read Replica	0.163	16.3 %

Джерело: розроблено автором

Перевірка узгодженості матриці критеріїв дала максимальне власне значення $\lambda_{\max} = 4.051$, індекс узгодженості $CI = 0.017$ та відношення узгодженості $CR = CI/RI = 0.019$ ($RI = 0.90$ для $n = 4$ [1]). Оскільки $CR = 0.019 < 0.10$, матрицю визнано узгодженою, а результати аналізу - коректними.

За підсумками аналізу першість із вагою 0.435 здобуло горизонтальне масштабування FastAPI (A4): продуктивність і масштабованість разом дають 75.5 % сумарної ваги, а саме за цими критеріями A4 має найвищі оцінки (0.542 і 0.514). Друге місце посідає Redis Cache (0.219) - найпростіше у впровадженні рішення, що дало найшвидший ефект на ранньому етапі; третє - Apache Kafka (0.182) з найвищою оцінкою надійності; четверте - PostgreSQL Read Replica (0.163) зі збалансованим, хоч і помірним, внеском за всіма критеріями.

Сформований рейтинг узгоджується з кількісними результатами повторного тестування: пропускна здатність системи зросла з вихідних 332 RPS (розділ 2) до 1 278.8 RPS після оптимізацій підрозділу 3.2 (приблизно у 3.9 рази), а після горизонтального масштабування, за розрахунками підрозділу 3.3, прогнозовано сягає близько 3 837 RPS - майже у 12 разів вище за базовий рівень за збереження якості обслуговування. Отже, упорядкування рішень за методом АНР відповідає їх фактичному внеску в підвищення продуктивності, масштабованості та надійності бекенд-системи.

ВИСНОВКИ

В кваліфікаційні роботі було визначено, як за допомогою моніторингу та математичного моделювання зробити мікросервісну бекенд-систему надійнішою і кращою до масштабування. Дослідження пройшло шлях від виявлення вузьких місць до їх усунення та перевірки результату повторним тестуванням, і нижче підсумовано здобуте на кожному з цих кроків.

Опрацювання літератури показало, що для застосунків розглянутого типу мікросервісний поділ виграє у можливості масштабувати окремі компоненти, проте переносить проблеми продуктивності на стик сервісів і мережеву взаємодію. У роботі зведено в єдину систему методи, якими ці проблеми розв'язують на практиці: балансування навантаження, репліки для читань, кеш за схемою Cache-Aside, винесення повільних операцій у чергу, пул з'єднань, індекси, а також захисні патерни обмеження конкурентності, запобіжника та швидкої відмови. Цей теоретичний апарат разом із набором метрик (час відповіді, перцентилі, RPS, частка помилок) далі застосовано як інструмент аналізу.

Аналітична частина почалася з UML-моделювання, яке дало три припущення щодо причин падіння продуктивності: брак вільних з'єднань до PostgreSQL, відсутність кешу для частих читань і затримка від синхронного звернення до SMTP. Навантажувальний експеримент у Locust із нарощуванням до 4 000 користувачів підтвердив усі три. Система впиралася у стелю 332 RPS, середній час відповіді тримався на рівні одинадцяти-дванадцяти секунд, а p95 сягав п'ятнадцяти-шістнадцяти. Окремо проявилися надлишкові ORM-звернення, відомі як проблема N+1, та незахищеність від сплесків конкурентних операцій.

Щоб ці чинники було видно вже в експлуатації, побудовано систему моніторингу на Prometheus, Grafana та Alertmanager зі збором показників одразу на трьох рівнях - застосунку, інфраструктури та зовнішніх залежностей. Перелік метрик і критичні пороги обрано за методологіями RED та USE, а правила сповіщень налаштовано так, щоб відрізнити ситуації для планової уваги від тих, що потребують негайної реакції. Завдяки автоматичним сповіщенням час виявлення

збою скорочується з годин до хвилин, а спостереження за тенденціями метрик дозволяє помічати проблему ще до її загострення.

Надійність піднято кількома узгодженими заходами. Кеш Redis для читаючих ендпоінтів за схемою Cache-Aside з TTL у 300 секунд зменшив медіанний час відповіді GET /api/projects з 11 577 до 910 мс, тобто майже у тринадцятеро. Відправлення листів перенесено у фоновий контур через Apache Kafka, через що POST /api/contact прискорився з 11 939 до 831 мс, приблизно у чотирнадцятеро, і вже без жодної помилки. Доповнили цей результат переписані ORM-вибірки зі складеними індексами, асинхронний пул asynsrg, семафор для обмеження важких операцій та запобіжник для зовнішніх сервісів; разом вони зняли базу даних із позиції головного вузького місця. Повторне тестування з тими самими параметрами зафіксувало системний приріст: пропускна здатність піднялася з 332 до 1 278.8 RPS, майже вчетверо, агрегований час відповіді опустився до 860 мс за медіаною та 2 100 мс за p95, а помилок не зафіксовано. Це означає, що в межах однієї конфігурації ресурс оптимізації використано повністю.

Подальші перспективи оцінено через теорію масового обслуговування. Модель M/M/1 за вимірної інтенсивності обслуговування близько 1 279 RPS відтворила саме той ефект, що спостерігався у тестах: коли коефіцієнт завантаження наближається до одиниці, час очікування в черзі зростає не лінійно, а різко. Звідси випливає, що одноінстансна схема вичерпала себе і потрібен перехід до кількох паралельних обслуговувачів. Відповідно запропоновано схему горизонтального масштабування зі stateless-інстансами FastAPI за балансувальником AWS ALB, паралельними воркерами в Kafka Consumer Group, окремою реплікою PostgreSQL під читання та автоматичним регулюванням кількості інстансів засобами AWS Auto Scaling. За розрахунком три репліки піднімають сукупну пропускну здатність до приблизно 3 837 RPS і повертають коефіцієнт завантаження з майже одиниці до значення близько 0.33, що в підсумку, разом із заходами оптимізації, дає близько дванадцятикратного зростання щодо стартового рівня без втрати якості обслуговування.

Запропоновані рішення були проаналізовані в третьому розділі за допомогою методу аналізу ієрархій за чотирма критеріями: продуктивністю, масштабованістю, надійністю та складністю впровадження. Найвищу оцінку отримало горизонтальне масштабування з вагою 0.435, далі йдуть Redis-кеш (0.219), Apache Kafka (0.182) і репліка PostgreSQL (0.163), а відношення узгодженості $CR = 0.019$ при допустимому значенні 0.10 свідчить, що судження несутеречливі. Розрахунок підтвердив, що ці рішення радше доповнюють одне одного, ніж конкурують, а їхні місця в рейтингу збігаються з реальним внеском у результат.

Отримані напрацювання доведено до конкретних конфігурацій і фрагментів коду, наведених у додатках, тож вони придатні до прямого впровадження в досліджувану систему. Сам хід роботи, від пошуку вузьких місць засобами тестування й моніторингу до їх усунення та підтвердження масштабування розрахунком, не прив'язаний до конкретного застосунку і переноситься на інші мікросервісні системи зі схожим, переважно читаючим профілем навантаження. Природним продовженням стане перевірка прогнозованих показників на реальному кластері та розширення моделі до мережі черг, яка враховувала б взаємний вплив кількох обслуговувачів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Комлева, Н. О. Поведінкове подання у форматі ХМІ в межах UML-метамоделі з двома поданнями. Вісник Хмельницького національного університету. Технічні науки, & Нікітченко, М. І. 2025 С.326–335. URL: <https://doi.org/10.31891/2307-5732-2025-357-42> (Дата звернення: 25.05.2026)
2. Литвинов, А. Л. Теорія систем масового обслуговування: навч. посіб. Харків: ХНУМГ ім. О. М. Бекетова. 2019. URL: <https://files.znu.edu.ua/files/Bibliobooks/Inshi60/0044257.pdf> (Дата звернення: 25.05.2026)
3. Улічев, О. С. Інноваційні рішення та переваги мікросервісної архітектури програмних продуктів. Центральнoукраїнський науковий вісник. Технічні науки, 10(41), Доренський, О. П., & Кулагін, В. П. 2024. С.16–29. URL: [https://mapiea.kntu.kr.ua/pdf/10\(41\)_I/4.pdf](https://mapiea.kntu.kr.ua/pdf/10(41)_I/4.pdf) (Дата звернення: 25.05.2026)
4. Що таке тестування навантаження. ZAPTEST. URL: <https://www.zaptest.com/uk/що-таке-тестування-навантаження> (Дата звернення: 25.05.2026)
5. Cloudflare. What is a cache hit ratio? URL: <https://www.cloudflare.com/learning/cdn/what-is-a-cache-hit-ratio/> (Дата звернення: 25.05.2026)
6. Connection pools and data sources. PostgreSQL JDBC Driver documentation. URL: <https://jdbc.postgresql.org/documentation/datasource/> (Дата звернення: 25.05.2026)
7. Fowler, M. Python concurrency with asyncio. Shelter Island, 2022. Manning Publications. (Дата звернення: 25.05.2026)
8. GeeksforGeeks. Cache-aside pattern. URL: <https://www.geeksforgeeks.org/system-design/cache-aside-pattern/> (Дата звернення: 25.05.2026)
9. Google. Find out how you stack up to new industry benchmarks for mobile page speed. Think with Google. 2017. URL: <https://business.google.com/ca->

[en/think/marketing-strategies/mobile-page-speed-new-industry-benchmarks/](https://en.think/marketing-strategies/mobile-page-speed-new-industry-benchmarks/) (Дата звернення: 25.05.2026)

10. Gregg, B. Systems performance: Enterprise and the cloud. Upper Saddle River, NJ: Prentice Hall. 2013. (Дата звернення: 25.05.2026)

11. Gregg, B. The USE method. 2013. URL: <https://www.brendangregg.com/usemethod.html> (Дата звернення: 25.05.2026)

12. Kleppmann, M. Designing data-intensive applications. Sebastopol, CA: O'Reilly Media. 2017. С.640. (Дата звернення: 25.05.2026)

13. Lewis, J. Microservices: A definition of this new architectural term. Fowler, M. 2014. URL: <https://martinfowler.com/articles/microservices.html> (Дата звернення: 25.05.2026)

14. Locust documentation. URL: <https://docs.locust.io/en/stable/what-is-locust.html> (Дата звернення: 25.05.2026)

15. Nath, K. All things caching: Use cases, benefits, strategies, choosing a caching technology, exploring some popular products. Medium. 2018. URL: <https://medium.com/datadriveninvestor/all-things-caching-use-cases-benefits-strategies-choosing-a-caching-technology-exploring-fa6c1f2e93aa> (Дата звернення: 25.05.2026)

16. Newman, S. Building microservices: Designing fine-grained systems (2nd ed.). Sebastopol: O'Reilly Media. С.616 . 2021. (Дата звернення: 25.05.2026)

17. NGINX. HTTP load balancing. URL: <https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/> (Дата звернення: 25.05.2026)

18. Ortega, J. M. Mastering Python for networking and security. Birmingham: Packt Publishing. С.586. 2023. (Дата звернення: 25.05.2026)

19. P95 vs P99 latency. URL: <https://loadtester.org/p95-vs-p99-latency> (Дата звернення: 25.05.2026)

20. Prometheus Authors. Overview. Prometheus Documentation. 2024. URL: <https://prometheus.io/docs/introduction/overview/> (Дата звернення: 25.05.2026)

21. Python Software Foundation. Global interpreter lock. Python Documentation. URL: <https://docs.python.org/3/glossary.html#term-global-interpreter-lock> (Дата звернення: 25.05.2026)

22. Richards, M. & Ford, N. Fundamentals of software architecture: An engineering approach. Sebastopol: O'Reilly Media. C.422. 2020. URL: <https://dokumen.pub/fundamentals-of-software-architecture-an-engineering-approach-1nbsped-1492043451-9781492043454.html> (Дата звернення: 25.05.2026)
23. Richardson, C. Microservices patterns: With examples in Java. Shelter Island: Manning Publications. C.520. 2018. (Дата звернення: 25.05.2026)
24. Richardson, C. Pattern: Microservice architecture. 2018. URL: <https://microservices.io/patterns/microservices.html> (Дата звернення: 25.05.2026)
25. Silberschatz, A., Korth, H. F. & Sudarshan, S. Database system concepts (7th ed.). New York: McGraw-Hill Education. C. 1344. 2019. URL: <https://www.mpgcamb.com/wp-content/uploads/2024/12/Abraham-Silberschatz-Henry-F.-Korth-S.-Sudarshan-Database-System-Concepts-McGraw-Hill-Education-2019.pdf> (Дата звернення: 25.05.2026)
26. Stopford, B. Designing event-driven systems. Sebastopol: O'Reilly Media. C.151. 2018. URL: <https://sitic.org/wordpress/wp-content/uploads/Designing-Event-Driven-Systems.pdf> (Дата звернення: 25.05.2026)
27. Understanding the cache-aside pattern: A practical guide. Medium. URL: <https://medium.com/@mallik-tech-vision/understanding-the-cache-aside-pattern-a-practical-guide-0c368bc71875> (Дата звернення: 25.05.2026)
28. What is load testing? URL: <https://loadtester.org/what-is-load-testing> (Дата звернення: 25.05.2026)
29. Wilkie, T. The RED method: How to monitor your microservices. GrafanaCon EU, Grafana Labs. 2018. URL: https://grafana.com/files/grafanacon_eu_2018/Tom_Wilkie_GrafanaCon_EU_2018.pdf (Дата звернення: 25.05.2026)
30. Average Response Time Calculation: How to Test Response Time. Dotcom-Monitor. 2026. URL: <https://www.dotcom-monitor.com/wiki/knowledge-base/average-response-time-calculat> (Дата звернення: 25.05.2026)

ДОДАТКИ

Додаток А

Сценарій навантажувального тестування

```
class GalaxyUser(HttpUser):
    host = "http://13.60.167.20:8000"
    wait_time = between(1, 3)
    project_ids = [1, 2, 3, 4, 5, 6]

    def on_start(self):
        self.client.headers.update({
            "Content-Type": "application/json",
            "Accept": "application/json",
        })
    @task
    def user_flow(self):
        with self.client.get(
            "/api/projects",
            name="/api/projects [list]",
            catch_response=True
        ) as resp:
            if resp.status_code == 200:
                resp.success()
            else:
                resp.failure(f"[projects] HTTP {resp.status_code}")
                logger.warning("Failed to load projects: %s",
resp.text[:120])
                return

        project_id = random.choice(self.project_ids)

        with self.client.get(
            f"/api/projects/{project_id}",
            name="/api/projects/{id} [detail]",
            catch_response=True
        ) as resp:
            if resp.status_code == 200:
                resp.success()
            else:
                resp.failure(f"[project/{project_id}] HTTP
{resp.status_code}")
                logger.warning("Failed to load project %d",
project_id)

# 3. Надсилання форми
if random.random() < 0.35:
    source = random.choice([
        "home", "projects", f"project_detail_{project_id}"])
```

```

        payload = {
            "name": f"Load Test User {random.randint(1000,
9999)}",
            "email": f"user{random.randint(1,
500)}@loadtest.com",
            "message": f"Inquiry about project {project_id} via
{source}",
            "source": source,
        }
    with self.client.post(
        "/api/contact",
        json=payload,
        name="/api/contact [form]",
        catch_response=True
    ) as resp:
        if resp.status_code in (200, 201):
            resp.success()
        else:
            resp.failure(f"[contact] HTTP
{resp.status_code}")
            logger.error("Contact form failed: %s",
resp.text[:120])

```

Додаток Б.1

Middleware для автоматичного збору

HTTP-метрик Prometheus

```
# app/core/middleware.py

class PrometheusMiddleware(BaseHTTPMiddleware):
    async def dispatch(self, request: Request, call_next) -> Response:
        path = self._normalize_path(request.url.path)
        method = request.method
        start_time = time.perf_counter()

        http_requests_in_progress.labels(method=method,
endpoint=path).inc()
        try:
            response = await call_next(request)
            status_code = response.status_code
            return response
        finally:
            duration = time.perf_counter() - start_time
            http_requests_total.labels(
                method=method, endpoint=path, status_code=str(status_code)
            ).inc()
            http_request_duration_seconds.labels(
                method=method, endpoint=path
            ).observe(duration)
            http_requests_in_progress.labels(method=method,
endpoint=path).dec()

    @staticmethod
    def _normalize_path(path: str) -> str:
        return "/".join(
            "{id}" if part.isdigit() else part
            for part in path.split("/")
        )
```

Додаток Б.2

Асинхронний декоратор кешування

відповідей (Cache-Aside, Redis)

```
import pydantic
import orjson
from functools import wraps
from redis.asyncio import Redis

def cache_response(ttl: int = 300):
    def decorator(func):
        @wraps(func)
        async def wrapper(*args, redis_client: Redis, **kwargs):
            cache_key = f"{func.__name__}:{str(args)}:{str(kwargs)}"
            cached_data = await redis_client.get(cache_key)
            if cached_data:
                return orjson.loads(cached_data)

            result = await func(*args, **kwargs)
            serialized = orjson.dumps(
                result.model_dump() if isinstance(result,
pydantic.BaseModel) else result
            )
            await redis_client.setex(cache_key, ttl, serialized)
            return result
        return wrapper
    return decorator
```

Додаток Б.3

Публікація події про відправлення листа в монік Kafka (FastAPI-продюсер)

```
import orjson
from fastapi import FastAPI, Depends
from aiokafka import AIOKafkaProducer

app = FastAPI()

async def get_kafka_producer() -> AIOKafkaProducer:
    producer = AIOKafkaProducer(bootstrap_servers='kafka:9092')
    await producer.start()
    try:
        yield producer
    finally:
        await producer.stop()

@app.post("/api/contact")
async def trigger_notification(payload: dict, producer:
AIOKafkaProducer = Depends(get_kafka_producer)):
    event_bytes = orjson.dumps({
        "event_type": "USER_NOTIFICATION",
        "recipient": payload["email"],
        "context": payload["meta"]
    })
    await producer.send_and_wait("telemetry-notifications",
event_bytes)
    return {"status": "accepted", "code": 202}
```

Додаток Б.4

Створення складеного індексу та оптимізована ORM-вибірка з усуненням проблеми N+1

```
from sqlalchemy import Index, String, DateTime, select
from sqlalchemy.orm import DeclarativeBase, Mapped, mapped_column,
joinedload

class Base(DeclarativeBase):
    pass

class Transaction(Base):
    __tablename__ = "transactions"
    id: Mapped[int] = mapped_column(primary_key=True)
    user_id: Mapped[int] = mapped_column(index=True)
    status: Mapped[str] = mapped_column(String(20))
    created_at: Mapped[DateTime] = mapped_column(DateTime)

    __table_args__ = (
        Index("idx_user_status_date", "user_id", "status",
"created_at"),
    )

    async def get_active_transactions(session, u_id: int):
        stmt = (
            select(Transaction)
            .where(Transaction.user_id == u_id, Transaction.status ==
"ACTIVE")
            .order_by(Transaction.created_at.desc())
            .execution_options(populate_existing=True)
        )
        result = await session.execute(stmt)
        return result.scalars().all()
```

Додаток Б.5

Налаштування асинхронного пулу

з'єднань SQLAlchemy/asyncpg

```
import asyncio
from sqlalchemy.ext.asyncio import create_async_engine, AsyncSession
from sqlalchemy.orm import sessionmaker
# URL підключення до хмарного інстансу AWS RDS PostgreSQL
DATABASE_URL = "postgresql+asyncpg://db_user:password@aws-rds-
instance.rds.amazonaws.com:5432/telemetry"

engine = create_async_engine(
    DATABASE_URL,
    pool_size=20,          # Фіксований ліміт сесій на один Docker-
контейнер
    max_overflow=10,       # Максимальне тимчасове перевищення
ліміту
    pool_timeout=30.0,     # Граничний час очікування сесії в черзі
пулу
    pool_recycle=1800,     # Автоматичне оновлення з'єднань для
запобігання stale connections в AWS
    pool_pre_ping=True     # Перевірка валідності сокета перед
виділенням корутині
)

AsyncSessionFactory = sessionmaker(
    bind=engine,
    class_=AsyncSession,
    expire_on_commit=False
)

async def get_db_session() -> AsyncSession:
    async with AsyncSessionFactory() as session:
        try:
            yield session
        finally:
            await session.close()
```

Додаток Б.6

Обмеження конкурентного виконання через `asyncio.Semaphore` з таймаутом

```
import asyncio
from fastapi import FastAPI, HTTPException, status

app = FastAPI()
heavy_task_semaphore = asyncio.Semaphore(5)

@app.get("/api/v1/reports/compute")
async def generate_heavy_report():
    try:
        async with asyncio.timeout(10.0):
            async with heavy_task_semaphore:
                # Імітація ресурсомісткої операції аналізу
                await asyncio.sleep(2.0)
                return {"status": "success", "data":
"metrics_report"}
    except asyncio.TimeoutError:
        raise HTTPException(
            status_code=status.HTTP_504_GATEWAY_TIMEOUT,
            detail="Server is heavily loaded. Try again later."
        )
```

Додаток Б.7

Запобіжник (Circuit Breaker) для захисту викликів до зовнішніх сервісів

```
import time
from fastapi import FastAPI, HTTPException, status
app = FastAPI()

class CircuitBreakerException(Exception): pass

class CircuitBreaker:
    def __init__(self, failure_threshold: int = 5, recovery_time: int
= 30):
        self.failure_threshold = failure_threshold
        self.recovery_time = recovery_time
        self.state = "CLOSED"
        self.failures = 0
        self.last_failure_time = 0

    def __call__(self):
        if self.state == "OPEN":
            if time.time() - self.last_failure_time >
self.recovery_time:
                self.state = "HALF-OPEN"
            else:
                raise CircuitBreakerException("Circuit breaker is
OPEN")

# Ініціалізація та утилізація патерну всередині обробника FastAPI
breaker = CircuitBreaker()

@app.get("/api/v1/external-sync")
async def external_integration_endpoint():
    try:
        breaker()
        return {"integration": "nominal"}
    except CircuitBreakerException:
        raise HTTPException(
            status_code=status.HTTP_503_SERVICE_UNAVAILABLE,
            detail="Circuit breaker active. Fast failing request.")
```

Додаток В.1

Конфігурація Docker Compose для розгортання трьох реплік FastAPI

```
# docker-compose.yml

api_service:
  image: backend-api:latest
  deploy:
    replicas: 3
    resources:
      limits:
        cpus: "2.0"
        memory: "1G"
    restart_policy:
      condition: on-failure
      max_attempts: 3
  command: gunicorn app.main:app -c gunicorn.conf.py
  healthcheck:
    test: ["CMD", "curl", "-f", "http://localhost:8000/health"]
    interval: 30s
    timeout: 10s
    retries: 3
```

Додаток В.2

Консюмер Kafka (Email Worker) із підтвердженням offset після обробки

```
# app/workers/email_worker.py

import asyncio
import orjson
from aiokafka import AIOKafkaConsumer
from app.services.smtp import smtp_service

async def run_email_worker() -> None:
    consumer = AIOKafkaConsumer(
        "contact_notifications",
        bootstrap_servers="kafka:9092",
        group_id="email-workers",          # спільна група →
        enable_auto_commit=False,          # offset підтверджується
        auto_offset_reset="earliest",
    )
    await consumer.start()
    try:
        async for msg in consumer:
            event = orjson.loads(msg.value)
            try:
                await smtp_service.send(
                    to=event["email"],
                    contact_id=event["contact_id"],
                )
                await consumer.commit()      # підтвердження лише
            except Exception:
                # offset не підтверджено → повідомлення буде
                # оброблено повторно (at-least-once)
                await asyncio.sleep(5)
    finally:
        await consumer.stop()
```

партиції діляться між воркерами вручну

після успішної відправки

Додаток В.3

Маршрутизація читання/запису через окремі engines (Primary/Replica)

```
# app/core/database.py

from sqlalchemy.ext.asyncio import create_async_engine, AsyncSession
from sqlalchemy.orm import sessionmaker

write_engine = create_async_engine(
    settings.DATABASE_WRITE_URL,
    pool_size=10, max_overflow=15, pool_pre_ping=True,
)
read_engine = create_async_engine(
    settings.DATABASE_READ_URL,
    pool_size=20, max_overflow=30, pool_pre_ping=True,
)

WriteSession = sessionmaker(write_engine, class_=AsyncSession)
ReadSession = sessionmaker(read_engine, class_=AsyncSession)

async def get_write_session() -> AsyncSession:
    async with WriteSession() as session:
        yield session

async def get_read_session() -> AsyncSession:
    async with ReadSession() as session:
        yield session
```

Додаток В.4

Політика автоматичного масштабування (AWS Auto Scaling Group)

```
# autoscaling-policy.yml

AutoScalingGroup:
  MinSize: 2
  MaxSize: 10
  DesiredCapacity: 3

ScaleOutPolicy:
  PolicyType: TargetTrackingScaling
  TargetTrackingConfiguration:
    PredefinedMetricType: ECSServiceAverageCPUUtilization
    TargetValue: 70.0
    ScaleOutCooldown: 180
    ScaleInCooldown: 600

StepScalingPolicy:
  StepAdjustments:
    - MetricIntervalLowerBound: 0
      MetricIntervalUpperBound: 100
      ScalingAdjustment: 1    # p95 > 400ms → +1
    - MetricIntervalLowerBound: 100
      ScalingAdjustment: 2    # p95 > 500ms → +2
```

Метадані

ДОКУМЕНТ

Заголовок

КЕР_КорнєєвВО

Автор

Корнєєв В О

Науковий керівник / Експерт

Агутін М.М.

ID документу

334356640

ОРГАНІЗАЦІЯ

Назва організації

Kyiv National Economic University named after Vadym Hetman
KNEU

Підрозділ

кафедра системного аналізу та кібербезпеки

ЗВІТ

Дата звіту

2026-06-18

Дата редагування

Обсяг знайдених подібностей

Коефіцієнт подібності визначає, який відсоток тексту по відношенню до загального обсягу тексту було знайдено в різних джерелах. Зверніть увагу, що високі значення коефіцієнта не автоматично означають плагіат. Звіт має аналізувати компетентна / уповноважена особа.



КП 1

25

Довжина фрази для коефіцієнта подібності 2



КП 2

13250

Кількість слів



КЦ

98896

Кількість символів

Тривога

У цьому розділі ви знайдете інформацію щодо текстових спотворень. Ці спотворення в тексті можуть говорити про МОЖЛИВІ маніпуляції в тексті. Спотворення в тексті можуть мати навмисний характер, але частіше характер технічних помилок при конвертації документа та його збереженні, тому ми рекомендуємо вам підходити до аналізу цього модуля відповідально. У разі виникнення запитань, просимо звертатися до нашої служби підтримки.

Заміна букв	Ⓜ	1
Інтервали	A→	0
Мікропробіли	␣	6
Білі знаки	␣	0
Парафрази (SmartMarks)	Ⓜ	17

Джерела

Нижче наведений список джерел. В цьому списку є джерела із різних баз даних. Колір тексту означає в якому джерелі він був знайдений. Ці джерела і значення Коефіцієнту Подібності не відображають прямого плагіату. Необхідно відкрити кожне джерело і проаналізувати зміст і правильність оформлення джерела.

10 найдовших фраз

Колір тексту

НАЗВА ТА АДРЕСА ДЖЕРЕЛА URL (НАЗВА БАЗИ)

КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ
(ФРАГМЕНТІВ)

1	Запара_Імплем. та адмін. кеш-систем для підв.пропускнуої здатності БД_курсова_ЗФМБ_123_денна(н.к.Шлурик Д.В) 5/7/2026 European University (European University)	12 0 (0.09 %)
2	https://www.mdpi.com/2227-7390/8/11/1873/html	12 0 (0.09 %)
3	https://ela.kpi.ua/server/api/core/bitstreams/d211d7a3-2b2e-4431-9bd8-b59f108ada6f/content	10 0 (0.08 %)
4	Запара_Імплем. та адмін. кеш-систем для підв.пропускнуої здатності БД_курсова_ЗФМБ_123_денна(н.к.Шлурик Д.В) 5/7/2026 European University (European University)	9 0 (0.07 %)
5	Дослідження ефективності багаторівневого кешування для підвищення продуктивності вебдодатків 6/3/2026 Taras Shevchenko National University of Kyiv (Факультет комп'ютерних наук та кібернетики)	9 0 (0.07 %)
6	Мушкаторов_дипломна_робота 6/3/2025 National Technical University of Ukraine Igor Sikorskyi Kyiv Politech Institute (ІАТЕ, К-ра цифрових технологій в енергетиці)	9 0 (0.07 %)
7	Запара_Імплем. та адмін. кеш-систем для підв.пропускнуої здатності БД_курсова_ЗФМБ_123_денна(н.к.Шлурик Д.В) 5/7/2026 European University (European University)	9 0 (0.07 %)
8	http://library.kpi.kharkov.ua/files/Vestniki/2018_16.pdf	9 0 (0.07 %)
9	Kozoriz_Karas_Diplom_2026 6/4/2026 V. N. Karazin Kharkiv National University (KKNU) (ІНІ комп'ютерних наук та штучного інтелекту - кафедра інтелектуальних програмних систем і технологій)	8 0 (0.06 %)
10	http://faculties.sbu.ac.ir/%7Eesadough/pdf/Advanced%20Engineering%20Mathematics%2010th%20Edition.pdf	8 0 (0.06 %)

База даних RefBooks (0.29 %)



#	ЗАГОЛОВОК	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)
---	-----------	---

джерело: RePEC

1	Coordination & cooperation in financial regulation: Do regulators comply with banking culture? Carretta, Alessandro,Schwizer, Paola,Farina, Vincenzo;	39 (7) (0.29 %)
---	--	-----------------

Домашня база даних



#	ЗАГОЛОВОК	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)
---	-----------	--

Програма обміну базами даних (0.87 %)



#	ЗАГОЛОВОК	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)
---	-----------	---

2	Запара_Імплем. та адмін. кеш-систем для підв.пропускнуої здатності БД_курсова_ЗФМБ_123_денна(н.к.Шлурик Д.В)	30 (3) (0.23 %)
---	---	-----------------

	5/7/2026 European University (European University)	
3	Дослідження ефективності багаторівневого кошування для підвищення продуктивності вебдодатків 6/3/2026 Taras Shevchenko National University of Kyiv (Факультет комп'ютерних наук та кібернетики)	21 (3) (0.16 %)
4	Kozoriz_Karas_Diplom_2026 6/4/2026 V. N. Karazin Kharkiv National University (KKNU) (ІНІ комп'ютерних наук та штучного інтелекту - кафедра інтелектуальних програмних систем і технологій)	15 (2) (0.11 %)
5	ФМФІТ магістр 2025 12/16/2025 Odessa I.I.Mechnikov National University (Одеський національний університет імені І.І.Мечникова)	13 (2) (0.1 %)
6	Шамиро_Дипломна_робота_на_платіт 11/7/2025 State University of Trade and Economics (Кафедра інженерії програмного забезпечення та кібербезпеки)	13 (2) (0.1 %)
7	Мушкаторов_дипломна_робота 6/3/2025 National Technical University of Ukraine Igor Sikorskyi Kyiv Politech Institute (ІАТЕ, К-ра цифрових технологій в енергетиці)	9 (1) (0.07 %)
8	RiabovaAS_MCS01_master_2025 8/12/2025 GoIT Neoversity (GoIT Neoversity)	7 (1) (0.05 %)
9	Малинко_дипломна_робота 6/6/2026 National Technical University of Ukraine Igor Sikorskyi Kyiv Politech Institute (ІАТЕ, К-ра цифрових технологій в енергетиці)	7 (1) (0.05 %)
Інтернет (0.63 %)		
#	Джерело URL	Кількість ідентичних слів (фрагментів)
10	http://library.kpi.kharkov.ua/files/Vestniki/2018_16.pdf	20 (3) (0.15 %)
11	https://www.mdpi.com/2227-7390/8/11/1873/html	18 (2) (0.14 %)
12	http://facufiles.sbu.ac.ir/%7Easadough/pdf/Advanced%20Engineering%20Mathematics%2010n%20Edition.pdf	15 (2) (0.11 %)
13	https://ocparchive.nure.ua/bitstreams/e5a1b3e7-5b62-488e-972d-440fb44e20e0/download	13 (2) (0.1 %)
14	https://ela.kpi.ua/server/api/core/bitstreams/d211d7a3-2b2e-4431-9bd8-b59f108ade6f/content	10 (1) (0.08 %)
15	https://studfile.net/preview/5592354/page-7/	7 (1) (0.05 %)

Список прийнятих фрагментів		
#	Зміст	Кількість однакових слів (фрагментів)

Копії тез доповідей та статей

Агутін М.М., Корнєєв В.О., Гребенюк А.Ю., “СМАРТ-АНАЛІЗ ПРОЦЕСІВ УПРАВЛІННЯ ВИМОГАМИ ТА ЗАБЕЗПЕЧЕННЯ ВІДМОВОСТІЙКОСТІ BACKEND-СИСТЕМ МОБІЛЬНИХ ДОДАТКІВ” збірник матеріалів тез IV конференції “Цифрова економіка” . Київ , 11-12 червня 2026.

Агутін М.М.,

к.е.н., доцент кафедри системного аналізу та кібербезпеки
КНЕУ імені Вадима Гетьмана

Гребенюк А.Ю.,

здобувач першого (бакалаврського) рівня вищої освіти,
КНЕУ імені Вадима Гетьмана

Корнєєв В.О.,

здобувач першого (бакалаврського) рівня вищої освіти,
КНЕУ імені Вадима Гетьмана

СМАРТ-АНАЛІЗ ПРОЦЕСІВ УПРАВЛІННЯ ВИМОГАМИ ТА ЗАБЕЗПЕЧЕННЯ ВІДМОВОСТІЙКОСТІ BACKEND-СИСТЕМ МОБІЛЬНИХ ДОДАТКІВ

Вступ

У сучасній IT-індустрії забезпечення високої доступності та відмовостійкості інформаційних систем є критично важливим аспектом кібербезпеки, що безпосередньо залежить від якості управління вимогами та стабільності backend-архітектури. Стрімке поширення мікросервісів і хмарних технологій вимагає впровадження інструментів смарт-аналітики для проактивного моніторингу метрик продуктивності та виявлення вразливостей систем під навантаженням. Недоліки на етапі проектування архітектури та хаотична зміна вимог призводять до деградації сервісів, виникнення внутрішніх загроз відмови в обслуговуванні (DoS) та фінансових збитків. У зв'язку з цим актуальним є дослідження методів смарт-аналізу процесів розробки та застосування захисних механізмів оптимізації backend-систем для підвищення їхньої загальної кіберстійкості.

Метою дослідження є підвищення кіберстійкості та відмовостійкості інформаційних систем шляхом розробки методів смарт-аналізу процесів управління вимогами у Scrum-командах та впровадження архітектурних захисних механізмів (кешування, асинхронності, fail-fast) для оптимізації продуктивності backend-платформ в умовах високого навантаження.