НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Навчально-науковий інститут атомної та теплової енергетики

Кафедра цифрових технологій в енергетиці

«До захисту допущено»
Завідувач кафедри
_____ Наталія АУШЕВА
«___»_____2026 р.

Дипломна робота
на здобуття ступеня бакалавр

За освітньою програмою «Цифрові технології в енергетиці»

Спеціальності 122 «Комп'ютерні науки»

на тему: «Програмна система балансування навантаження зворотного проксі-серверу»

Виконав: студент 4 курсу, групи ТР-23
Малярчук Богдан Васильович _____
(прізвище, ім'я, по батькові) (підпис)

Керівник доцент кафедри ЦТЕ, к.т.н. Анатолій ДЕМЧИШИН _____
(посада, науковий ступінь, вчене звання, ім'я, ПРІЗВИЩЕ) (підпис)

Рецензент доцент кафедри ІПЗЕ, к.т.н. Іван ВАРАВА___ _____
(посада, науковий ступінь, вчене звання, ім'я, ПРІЗВИЩЕ) (підпис)

Н.контроль ас. Артем МЕЛЬНИЧЕНКО _____
(посада, ім'я, ПРІЗВИЩЕ) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2026

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ АТОМНОЇ ТА ТЕПЛОВОЇ ЕНЕРГЕТИКИ

Кафедра ЦИФРОВИХ ТЕХНОЛОГІЙ В ЕНЕРГЕТИЦІ

Рівень вищої освіти – перший (бакалаврський)
спеціальність 122 «Комп'ютерні науки»
Освітньо-професійна програма «Цифрові технології в енергетиці»

ЗАТВЕРДЖУЮ
Завідувач кафедри ЦТЕ
_____ Наталія АУШЕВА
(підпис)
«___» _____2026 р.

ЗАВДАННЯ

на дипломну роботу студенту

_____Малярчуку Богдану Васильовичу_____
(прізвище, ім'я, по батькові)

1. Тема роботи «Програмна система балансування навантаження зворотного проксі-серверу»
Науковий керівник Демчишин Анатолій Анатолійович, к.т.н.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від « 28 » травня 2026 року № 1992 -с

2. Термін подання студентом роботи 03.06.2026

3. Вихідні дані до роботи мова програмування Python, фреймворк React, СУБД SQLite, VM VirtualBox – Ubuntu Mate 22.04

4. Перелік питань, які потрібно розробити _____

1) провести огляд існуючих систем балансування;
2) обрати інструменти реалізації програмної системи;

3) розробити архітектуру і реалізувати систему конфігурації та моніторингу мультисервісної вебплатформи конвертації форматів медіафайлів на основі зворотного проксі-сервера;
4) порівняти різні стратегії балансування навантаження й визначити найбільш ефективну для сервісів конвертації форматів.
5. Орієнтований перелік ілюстративного матеріалу діаграма використання програмної системи «Балансування навантаження зворотного проксі-серверу», ER-діаграма бази даних, знімки екрану роботи користувачів з вебзастосунком.
6. Дата видачі завдання 15.09.2025

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1.	Вибір теми роботи	02.06.2025	виконано
2.	Аналіз предметної області та огляд існуючих рішень	03.06.2025 – 15.09.2025	виконано
3.	Розробка архітектури системи	01.10.2025 – 30.11.2025	виконано
4.	Програмна реалізація системи	01.12.2025 – 31.03.2026	виконано
5.	Розробка окремих підсистем	01.04.2026 – 15.05.2026	виконано
6.	Тестування і налагодження програмної системи	16.05.2026 – 25.05.2026	виконано
7.	Оформлення пояснювальної записки	протягом усього періоду	виконано
8.	Захист програмного забезпечення	12.05.26	виконано
9.	Передзахист	02.06.26	виконано
10.	Захист	17.06.26	виконано

Студент _____ Богдан МАЛЯРЧУК
(підпис) (прізвище та ініціали)

Керівник _____ Анатолій ДЕМЧИШИН
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Дипломна робота виконана на 76 сторінках, містить 24 ілюстрації, 2 додатки, 21 джерело в переліку посилань.

Мета роботи – розробити програмну систему для балансування навантаження мультисервісної вебплатформи на основі зворотного проксі-сервера.

Методи та засоби: стратегії балансування навантаження, алгоритми реалізації сервісів конвертації, мова програмування Python, вебфреймворк React, база даних SQLite.

Результат – програмна система балансування навантаження вебсервісів, розгорнутих у Docker-контейнерах.

Ключові слова: стратегії балансування, сервіси конвертації, гетерогенне навантаження.

ABSTRACT

The thesis is 76 pages long, contains 24 figures, 2 appendixes, and 21 references.

The aim of the thesis is to develop a software system for load balancing a multi-service web platform based on a reverse proxy server.

Methods and tools: load balancing strategies, algorithms for implementing conversion services, the Python programming language, the React web framework, and the SQLite database.

The result is a software load balancing system for web services deployed in Docker containers.

Keywords: load balancing strategies, conversion services, heterogeneous workload.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

API (Application Programming Interface) – програмний інтерфейс взаємодії між окремими компонентами програмної системи або між різними системами через стандартизовані запити та відповіді.

CRUD (Create, Read, Update, Delete) – чотири базові операції роботи з даними в реляційних базах даних: створення, читання, оновлення та видалення записів.

ERD (Entity Relationship Diagram) – діаграма сутностей і зв’язків, що відображає структуру бази даних та взаємозв’язки між її таблицями.

ПС – програмна система.

Гетерогенність навантаження – навантаження на систему, при якому різні запити суттєво відрізняються між собою за часом обробки та споживанням ресурсів процесора. Наприклад, конвертація PDF-файлів вимагає значно більше ресурсів, ніж конвертація зображень.

Контейнеризація – технологія ізоляції застосунку разом із усіма його залежностями в окремому середовищі виконання – контейнері – без необхідності встановлення повноцінної операційної системи.

Оркестрація – автоматизоване керування життєвим циклом контейнерів: їх запуск, зупинка, масштабування та відновлення після збоїв залежно від поточного стану системи.

ВСТУП

У сучасних умовах зростає залежність користувачів від вебсервісів для виконання типових операцій, зокрема конвертації мультимедійних файлів (зображень, аудіо, тощо). У зв’язку з цим серверна частина таких систем повинна забезпечувати обробку значної кількості одночасних запитів при збереженні стабільного й мінімального часу відгуку, що безпосередньо впливає на якість користувацького досвіду та загальну продуктивність системи.

Розробка системи, що включає в себе кілька сервісів конвертації під єдиною точкою доступу, вимагає обґрунтованого вибору стратегії балансування навантаження. Різні сервіси мають суттєво різні обчислювальні вимоги: наприклад, конвертація PDF-файлів є найбільш ресурсоємною для процесора, тоді як обробка аудіоформатів потребує середніх обчислювальних ресурсів, а конвертація зображень – мінімальних. Така гетерогенність навантаження створює складності для рівномірного розподілу запитів і вимагає адаптивного підходу до балансування ресурсів.

Мета дипломної роботи – розробити програмну систему для балансування навантаження мультисервісної вебплатформи на основі зворотного проксі сервера.

Для досягнення поставленої мети необхідно вирішити такі завдання:

— провести огляд існуючих систем балансування;

— обрати інструменти реалізації програмної системи;

— розробити архітектуру і реалізувати систему конфігурації та моніторингу мультисервісної вебплатформи конвертації форматів медіафайлів на основі зворотного проксі серверу;

— порівняти різні стратегії балансування навантаження й визначити найбільш ефективну для сервісів конвертації форматів.

Апробація результатів дипломної роботи проведено на XXIII міжнародній науково-практичній конференції молодих вчених та студентів «Сучасні проблеми наукового забезпечення енергетики: безпека, стійкість, ІТ та екологічний моніторинг» (до 40-річчя Чорнобильської катастрофи) [1]. Додатковим підтвердженням автентичності публікації є додаток Б, у якому розміщено знімки сторінок публікації.

Система складається з кількох координованих компонент:

— конфігураційний сервер з графічним вебінтерфейсом для моніторингу та конфігурації пулів, контейнерів і машин;

— набір активних Docker-образів, розміщених у hub.docker.com репозиторії;

— головна сторінка із набором сервісів конвертації;

— навантажувальний k6-скрипт;

— сценарії тестування продуктивності.

Обсяг текстової частини роботи складає 56 сторінок.

Основна частина дипломної роботи охоплює чотири розділи, у яких послідовно описується весь процес розробки програмної системи – від формулювання задачі до демонстрації готового продукту в роботі.

У першому розділі наводиться постановка прикладної задачі, яку розв’язує система, робиться огляд існуючих стратегій балансування навантаження та програмних засобів, обраних для реалізації. Такий підхід до структурування роботи відповідає загальноприйнятій практиці опису програмних систем, за якою документація повинна включати в себе як задачу, так і засоби її вирішення. Стандарт IEEE 1471 визначає практики для формування системи понять і фреймворку при описі програмної архітектури. Саме тому перший розділ є фундаментальним для всієї подальшої роботи [2].

Другий розділ присвячений детальному опису засобів розробки програмного забезпечення. Описуються такі засоби, як:

— мова програмування Python та її бібліотеки;

— база даних SQLite;

— бібліотека React для побудови інтерфейсу;

— інструменти Docker та Docker Compose для контейнеризації;

— NGINX як зворотний проксі та вебсервер;

— утиліта curl для тестування;

— інструмент навантажувального тестування k6.

Обґрунтовано вибір кожного засобу та його роль у системі.

Третій розділ містить опис архітектури розробленої програмної системи. Тут наводиться структура проєкту, описуються та детально розглядаються критичні компоненти системи та взаємодія між ними:

— сторінка адміністратора з усіма вкладками для моніторингу і конфігурації;

— сторінка звичайного користувача;

— реляційна база даних із поясненням кожної таблиці.

Четвертий розділ описує роботу користувача з програмною системою. Тут покроково показано, як користувач взаємодіє з вебплатформою конвертації форматів файлів – від вибору сервісу до отримання готового конвертованого файлу.

Разом ці чотири розділи утворюють повний і послідовний опис розробленої програмної системи – від постановки задачі до підтвердження досягнутих результатів.

1 ОСОБЛИВОСТІ СИСТЕМИ БАЛАНСУВАННЯ НАВАНТАЖЕННЯ ЗВОРОТНОГО ПРОКСІ-СЕРВЕРА

Балансування навантаження зворотного проксі серверу широко вивчається та впроваджується для оптимізації надійності та відмовостійкості вебсервісів. В умовах стрімкого зростання інтернет-трафіку та підвищених вимог користувачів до швидкості відповіді, ефективний розподіл навантаження між серверами став критичною складовою сучасної вебархітектури. Дослідження показують, що розгорнуті на хмарних платформах мікросервіси суттєво залежать від конфігурації балансувальників навантаження, а їх неоптимальне налаштування може негативно впливати на продуктивність всієї системи [3].

1.1. Постановка прикладної задачі

У даній дипломній роботі розглядається задача проєктування та реалізації системи балансування навантаження на основі зворотного проксі-сервера для мультисервісної вебплатформи конвертації форматів файлів. Розроблювана система має забезпечувати одночасне обслуговування багатьох користувачів з дотриманням вимог до часу відповіді та доступності сервісів.

Розроблена система – це система балансування навантаження на основі зворотного проксі-сервера, що перенаправляє клієнтські запити до різних розподілених backend-сервісів використовуючи NGINX як зворотний проксі, поведінка маршрутизації якого керується через контрольовані параметри.

Зворотний проксі-сервер виступає єдиною точкою входу для клієнтських запитів і перенаправляє їх до відповідних серверів відповідно до обраної стратегії балансування навантаження. Оптимальна стратегія забезпечує рівномірний розподіл навантаження між екземплярами сервісів із мінімальною затримкою.

1.2. Огляд стратегій балансування навантаження

Стратегія балансування навантаження визначає, за яким правилом черговий запит буде направлено на один із доступних серверів. Вибір стратегії суттєво впливає на швидкість відповіді системи та рівномірність розподілу навантаження між серверами [4]. Не існує універсально кращого алгоритму балансування – вибір залежить від самого застосунку: чи є запити однорідними чи різними за тривалістю, чи важлива прив’язка сесії користувача до конкретного сервера [5].

У даній роботі реалізовано та порівняно п’ять стратегій балансування навантаження.

Round Robin – найбільш простий та найбільш поширений алгоритм. Запити розподіляються по черзі між серверами у циклічному порядку: перший запит іде на перший сервер, другий – на другий, і так по колу. Головна перевага Round Robin полягає у передбачуваній поведінці та мінімальних вимогах до ресурсів. Однак алгоритм не враховує поточне навантаження серверів та різну тривалість обробки запитів.

Least Connections – адаптивний алгоритм, який направляє новий запит на сервер з найменшою кількістю активних з’єднань у поточний момент. Такий підхід до балансування навантаження є більш інтелектуальним, ніж Round Robin, особливо коли запити суттєво відрізняються за тривалістю обробки. Для платформи конвертації файлів, де PDF-конвертація значно важча за конвертацію зображень, цей алгоритм є особливо доречним.

IP Hash – алгоритм, при якому сервер обирається на основі хешу IP-адреси клієнта. Це гарантує, що запити від одного клієнта завжди потрапляють на один і той самий сервер, що корисно для підтримки сесій користувача.

Random характерний тим, що запит направляється на випадково обраний сервер. При великій кількості запитів випадковий алгоритм забезпечує рівномірний розподіл між серверами, хоча і не враховує їхній поточний стан. Іноді використовуються ваги – тоді деякі вузли стають більш потужними, що стає сигналом для алгоритму про їх пріоритетність.

Power of Two – компромісний алгоритм між Random та Least Connections. Випадково обираються два сервери, і з них обирається менш завантажений. Це дозволяє уникнути як повного ігнорування стану серверів (як-от у Random), так і необхідності перевіряти всі сервери (як у Least Connections).

1.3. Огляд програмних засобів

Серед відомих балансувальників навантаження виділяють такі:

— Nginx – один з найбільш поширених вебсерверів і зворотних проксі, що підтримує балансування методами Round Robin, Least Connections та IP Hash;

— HAProxy – спеціалізоване рішення для балансування навантаження із детальним моніторингом і гнучким налаштуванням;

— Traefik – динамічний зворотний проксі, розроблений для сучасних контейнеризованих середовищ з вбудованою підтримкою Docker та Kubernetes).

Хмарні сервіси на кшталт AWS Elastic Load Balancer та Azure Application Gateway автоматично масштабують ресурси та перевіряють доступність сервісів.

У даній роботі балансувальник реалізовано власноруч мовою Python з використанням фреймворку FastAPI, що дозволило гнучко реалізувати всі п’ять алгоритмів балансування. Існуючі рішення на зразок NGINX підтримують лише обмежений набір алгоритмів і не надають можливості програмно фіксувати метрики кожного запиту з прив’язкою до алгоритму.

Технології контейнеризації відкрили нові можливості для розгортання застосунків в різних обчислювальних середовищах. Docker-контейнери не включають повноцінну операційну систему, натомість використовують спільне ядро хост-машини, що робить їх значно легшими та ефективнішими порівняно з віртуальними машинами [6]. Docker-контейнери вимагають менше апаратних ресурсів, ніж віртуальні машини, і мають значно швидший час запуску, що робить їх ідеальними для середовищ з частими циклами розробки та потребою у масштабуванні [7]. Це робить контейнери особливо привабливими для горизонтального масштабування при балансуванні навантаження [8].

Для управління екосистемою контейнерів використовується Docker Compose, який дозволяє описати всю інфраструктуру системи в одному конфігураційному файлі та запустити її однією командою. Для автоматичного масштабування контейнерів залежно від поточного навантаження на CPU розроблено bash-скрипт, який запускає нові екземпляри серверів при перевищенні порогового значення завантаженості. Для тестування продуктивності системи використовується інструмент навантажувального тестування k6 від Grafana Labs, який дозволяє моделювати роботу великої кількості багатьох одночасних користувачів і збирати метрики латентності та пропускної здатності для кожного з алгоритмів балансування.

2 ЗАСОБИ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

У процесі розробки програмного продукту було застосовано сучасні технології, вибір яких обумовлений потребою універсальності та високої здатності до супроводу програмного забезпечення. Далі наведено опис основних засобів розробки, використаних у межах створення програмної системи балансування навантаження зворотного проксі-сервера.

2.1. Python

Python було обрано як основну мову реалізації серверної частини – застосунку балансувальника навантаження.

Мова має особливості, які є вирішальними для ефективної розробки програмного забезпечення:

— розвинена екосистема бібліотек;

— інтегрованість з інструментами контейнеризації та тестування;

— широко документована і підтримувана спільнотою.

За підрахунком платформи GitHub, віддалений репозитарій програмного продукту містить близько 36% коду мовою Python від загального обсягу коду проєкту, що підтверджує її центральну роль у розробці системи (рис. 2.1).

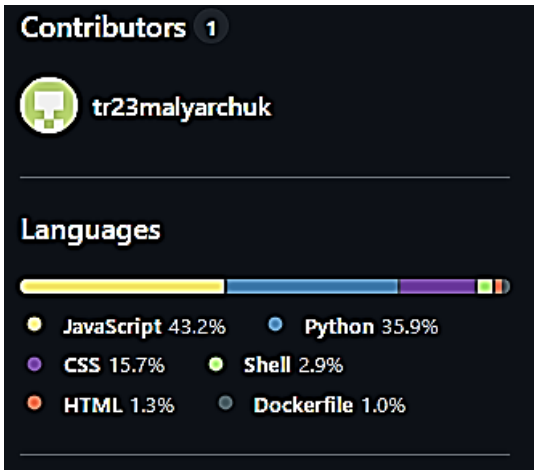


Рисунок 2.1 – Відсотковий розподіл обсягу коду програмного продукту між різними мовами програмування

У межах проєкту Python використовується у кількох ключових компонентах системи. Балансувальник навантаження реалізовано за допомогою фреймворку Fast API, який дозволяє будувати асинхронні вебзастосунки. Асинхронність означає, що сервер може одночасно обробляти багато запитів від різних користувачів, не чекаючи завершення кожного з них по черзі. Це особливо важливо для платформи конвертації файлів, де кожен запит може тривати від кількох секунд до кількох хвилин [9].

Для взаємодії з Docker-сокетом використовується бібліотека httpx, яка дозволяє відправляти асинхронні HTTP-запити. Саме через неї балансувальник кожні 10 секунд запитує у Docker список запущених контейнерів і оновлює пул доступних серверів. Якщо контейнер зупинився, він автоматично виключається з пулу, і нові запити на нього більше не направляються.

Для зберігання даних про кожен оброблений запит використовується вбудована бібліотека sqlite3, яка дозволяє працювати з базою даних SQLite без встановлення окремого сервера.

2.2. SQLite

SQLite – компактна та продуктивна реляційна база даних, яка використовується для зберігання даних моніторингу. Завдяки відсутності необхідності в окремому серверному процесі та мінімальним вимогам до конфігурації, SQLite забезпечує високий рівень надійності з простим механізмом розгортання в межах розробленої програмної системи.

SQLite в контексті розробленої програмної системи зберігає всю базу даних в одному файлі, що значно спрощує її перенесення між середовищами розробки та експлуатації. Це робить її особливо зручною для невеликих та середніх проєктів, де не потрібно складна серверна інфраструктура, але є досить обмеженим вибором для складних проєктів.

База даних підтримує стандарт SQL, що дозволяє використовувати звичні оператори для створення, оновлення та вибірки даних. На відміну від MySQL, яка вимагає налаштування серверного процесу та мережевих підключень, SQLite забезпечує простіший механізм розгортання, хоча її можливості масштабування та паралельної обробки запитів обмежені.

З точки зору продуктивності, SQLite ефективно працює з локальними запитами, забезпечуючи швидкий доступ до даних без затримок у мережі. Крім того, вона підтримує транзакційність, що гарантує цілісність у випадку збоїв або аварійного завершення роботи програми.

Таким чином, SQLite знижує складність розробки, оскільки не потребує адміністрування окремого сервера баз даних. SQLite є доцільними вибором для системи моніторингу балансування навантаження, де важливими є простота, стабільність та мінімальні витрати ресурсів [10].

2.3. React

React – декларативна JavaScript-бібліотека для побудови користувацьких інтерфейсів. У межах розробленого програмного продукту React застосовується для реалізації клієнтської частини застосунку. Компонентна архітектура бібліотеки забезпечує модульність і повторне використання елементів інтерфейсу, що спрощує супровід та масштабування фронтенд-частини системи. Середовище виконання Node.js слугує платформою для збірки та локального розгортання React-застосунку, що надає доступ до менеджера пакетів npm і необхідного інструментарію розробки.

В рамках проєкту було використано можливість React щодо відкладеного завантаження компонентів – React.lazy. Адмін-панель є окремим компонентом, який за замовчування не завантажується разом із основним застосунком. Її відображення керується слайдером «Developer Mode» (рис. 2.2) на сторінці користувача.

Рисунок 2.2 – Стани слайдера «Developer Mode»

Поведінка слайдера визначена наступним чином:

— коли слайдер переведено у стан ON, компонент адмін-панелі завантажується динамічно;

— коли OFF – він не рендериться.

Такий підхід зменшує обсяг коду, який браузер завантажує при першому ж відкритті сторінки, і є рекомендованою практикою, згаданою в офіційній документації фреймворку React [11].

Вебплатформа складається з двох головних компонентів – App та AdminPanel, кожен з яких реалізує окрему сторінку застосунку.

Компонент App є сторінкою звичайного користувача: він відображає чотири різнокольорові плитки сервісів конвертації. Користувач має можливість обрати будь-яку з цих плиток, обрати файл із відповідним розширенням і надіслати його на обробку. Результат автоматично завантажується у браузері, а якщо ні – можна на натиснути на кнопку «Завантажити оброблений файл», що з’явиться одразу після успішної конвертації файлу, надаючи додаткову можливість завантажити конвертований файл за тієї умови, якщо автоматичне завантаження не відбудеться.

Компонент AdminPanel – це панель адміністратора з можливістю моніторингу навантаження на сервери та конфігурації існуючих пулів, інстансів й машин. Панель адміністратора доступна лише у режимі розробника, який можна активувати слайдером «Developer Mode».

Для керування станом інтерфейсу в обох компонентах використовується хук useState, який є стандартним інструментом React для зберігання змінних даних усередині компонента. Наприклад, у компоненті App через useState зберігаються:

— обраний сервіс;

— завантажений файл;

— стан завантаження;

— посилання на результат.

Хук useCallback застосовується для запобігання зайвому повторному рендерингу при передачі функцій як параметрів дочірнім компонентам [12].

В компоненті AdminPanel реалізовано власний хук usePoll, який автоматично повторює запит до API через заданий інтервал часу за допомогою setInterval. Це дозволяє адмін-панелі оновлювати дані про запущені сервери, навантаження на контейнери та статистику запитів кожні кілька секунд без ручного оновлення сторінки.

Хук useEffect використовується для запуску опитування при першому рендері компонента і його зупинки при видаленні компонента зі сторінки, що запобігає витокам пам’яті.

Графіки в адмін-панелі реалізовані за допомогою вбудованого SVG – без використання сторонніх бібліотек для побудови діаграм. Кожен стовпець гістограми малюється як SVG-елемент з динамічно обчисленою висотою, а підписи значень розміщуються над стовпцями. Перевагою такого підходу є цілковита незалежність від зовнішніх пакетів для візуалізації та зменшення розміру фінального програмного продукту.

Вебзастосунок зібраний за допомогою команди create-react-app і упакований у Docker-контейнер разом із вебсервером NGINX, який роздає статичні файли і проксіює API-запити до балансувальника. Завдяки цьому користувач взаємодіє лише з одним портом – 80, а всі запити до бекенду передаються прозоро через NGINX без необхідності знати адресу балансувальника.

Такий підхід забезпечує просте масштабування фронтенду: нові екземпляри React-застосунку можуть бути швидко розгорнуті на різних серверах без зміни налаштувань клієнта. Також реалізована централізована обробка помилок і повідомлень від серверної частини, що дозволяє адміністраторам і користувачам своєчасно отримувати інформацію про статус системи та відмови сервісів.

2.4. curl

Використання curl як HTTP-клієнта для ручного тестування та діагностики взаємодії між компонентами системи дозволяє заощадити час на вирішення потенціальних неполадок. Інструмент надає зручний інтерфейс командного рядка для формування тестових HTTP-запитів і аналізу відповідей сервера, що робить його незамінним у Linux-дистрибутивах. Додатково, застосування curl дає можливість перевіряти коректність роботи REST API без необхідності використання графічних клієнтів, що спрощує troubleshooting-процес та прискорює цикл розробки.

Утиліта curl підтримує всі основі HTTP-методи (GET, POST, PUT, DELETE), що дозволяє перевіряти CRUD-операції у REST-сервісах без написання додаткового коду. Така гнучкість особливо корисна при troubleshooting-процесі мікросервісів або балансувальника навантаження, де потрібно швидко оцінити поведінку окремих точок доступу. На відміну від браузерів, curl надає детальний контроль над заголовками, тілом запиту та параметрами аутентифікації, що є важливим при тестуванні захищених API.

Інструмент розповсюджується як вільне та відкрите програмне забезпечення, сумісне з багатьма платформами, що робить його універсальним для використання інженерами будь-якої технологічної області. Використання curl як частини тестування допомагає зменшити залежність проєкту від графічних інструментів, підвищує прозорість мережевої взаємодії та сприяє швидшому виявленню логічних чи конфігураційних проблем мережі [13].

2.5. Docker

Docker використовується як платформа контейнеризації, що забезпечує ізоляцію компонентів системи та відтворюваність середовища виконання незалежно від конфігурації цільової інфраструктури. Кожен сервіс розробленої системи упаковано в окремий контейнер із можливістю їх подальшого розміщення на Docker Hub, що спрощує процедуру розгортання.

У межах системи кожен серверний екземпляр (srv1-srv4) запускається як окремий контейнер, що дозволяє гнучко керувати їх життєвим циклом (рис. 2.3). Це дає можливість динамічно створювати та зупиняти сервіси без впливу на інші компоненти системи. Використання ізольованих контейнерів зменшує ризик конфліктів залежностей між програмними середовищами.

Рисунок 2.3 – Запуск екземплярів серверів як контейнерів

У представленому рішенні Docker також використовується для організації внутрішньої мережі контейнерів, що забезпечує їх взаємодію через іменовані хости. Завдяки цьому спрощується маршрутизація запитів між балансувальником навантаження та бекенд-серверами.

Скрипт оркестрації контейнерів дозволяє автоматично запускати, зупиняти та видаляти екземпляри залежно від навантаження.

Отже, Docker підвищує гнучкість системи, спрощує її розгортання та забезпечує стабільну роботу в різних середовищах виконання [14].

2.6. Docker Compose

Docker Compose застосовується для оркестрації екосистеми Docker-контейнерів. Він визначає залежності між сервісами, налаштовує мережеву взаємодію і керує життєвим циклом контейнерів через єдиний конфігураційний файл у форматі YAML [15]. Завдяки цьому розробник описує всю інфраструктуру системи в одному місці, а запуск усіх компонентів зводиться до однієї команди – docker compose up.

У розробленій системі файл docker-compose.yml описує два основних сервіси:

— backend-сервіс – це балансувальник навантаження, зібраний з директорії /backend. Він запускається у контейнері з іменем lb і слухає порт 8000;

— frontend-сервіс – це React-застосунок разом із вбудованим вебсервером NGINX, зібраний з директорії /frontend. Він доступний на порту 80 і залежить від backend, тобто Docker Compose гарантує, що балансувальник буде запущений першим.

Особливу роль у конфігурації відіграє монтування Docker-сокету. Рядок /var/run/docker.sock:/var/run/docker.sock у секції volumes дозволяє балансувальнику зсередини свого контейнера звертатися до Docker-деймона хост-машини. Це необхідно для того, щоб DynamicPool у main.py (балансувальнику) міг отримувати список запущених контейнерів srv1-srv4 без встановлення будь-яких додаткових інструментів.

Без цього монтування балансувальник не зміг би динамічно визначити, які сервери зараз доступні.

Директива extra_hosts: host.docker.internal:host-gateway вирішує задачу мережевої адресації між контейнером і хост-машиною. Вона дозволяє балансувальнику звертатися до контейнерів srv1-srv4, які запущені не через Docker Compose, а через скрипт масштабування контейнерів безпосередньо на хості. Таким чином, Docker Compose і bash-скрипт масштабування працюють у єдиній мережевій екосистемі, не заважаючи одне одному.

Обидва сервіси підключені до спільної мережі lb-net з драйвером bridge. Це означає, що frontend і backend можуть звертатися один до одного за іменем сервісу. Наприклад, NGINX у frontend-контейнері перенаправляє запити на адресу host.docker.internal:8000, де слухає балансувальник. Директива restart: unless-stopped забезпечує автоматичний перезапуск обох контейнерів після перезавантаження системи або у разі несподіваного завершення процесу.

Директорія ./data монтується у контейнер за шляхом /app/data, де зберігається файл бази даних requests.db. Відповідно, всі дані про оброблені запити зберігаються на хост-машині і не зникають після зупинки або перезапуску контейнера. Такий підхід забезпечує збереження статистики між сесіями тестування навантаження, що є критичним для порівняльного аналізу алгоритмів балансування.

Отже, Docker Compose суттєво знижує операційну складність розгортання системи, а вся інфраструктура запускається і зупиняється однією командою, що робить систему зручною як для локальної розробки, так і для демонстрації результатів дипломної роботи (додаток А).

2.7. NGINX

NGINX – це високоефективне програмне забезпечення з відкритим вихідним кодом, яке виконує функції вебсервера, балансувальника навантаження та зворотного проксі-сервера, забезпечуючи HTTP-маршрутизацію та кешування відповідей, що дозволяє знизити навантаження на прикладний рівень і підвищити загальну пропускну здатність системи. NGINX як зворотний проксі-сервер приймає запити від користувачів та перенаправляє їх на внутрішній сервер Node.js-додатку, приховуючи реальну архітектуру мережі.

Використання NGINX дозволяє ефективно розподіляти вхідні HTTP-запити між кількома серверними екземплярами, що підвищує відмовостійкість системи. Завдяки асинхронній архітектурі NGINX здатний обробляти велику кількість одночасних з’єднань із мінімальним використанням ресурсів. У результаті його застосування покращується стабільність роботи системи та зменшується затримка при обробці запитів користувачів [16].

2.8. k6

k6 – інструмент навантажувального тестування з відкритим вихідним кодом, розроблений компанією Grafana Labs, орієнтований на тестування HTTP-інтерфейсів і вебзастосунків. Особливості k6 полягають у наступному:

— перевірка застосунків та інфраструктури на надійність та продуктивність;

— допомога командам у запобіганні помилкам та порушенням на рівні обслуговування;

— створення стійких систем, здатних витримувати високі та надвисокі навантаження.

3 ОПИС АРХІТЕКТУРИ СИСТЕМИ БАЛАНСУВАННЯ НАВАНТАЖЕННЯ НА ОСНОВІ ЗВОРОТНОГО ПРОКСІ-СЕРВЕРА

Програмна система балансування навантаження зворотного проксі-серверу складається з кількох взаємопов’язаних компонентів:

— вебдодатку;

— серверної частини;

— сервісів конвертації файлів;

— оркестратора контейнерів;

— зворотного проксі-сервера.

Центральним елементом архітектури є NGINX, який виконує роль зворотного проксі-сервера, єдиної точки входу та балансувальника навантаження. Усі вхідні HTTP-запити від клієнтів надходять спочатку до NGINX, який розподіляє їх між активними серверами відповідно до поточного навантаження. Такий підхід забезпечує горизонтальне масштабування системи та рівномірний розподіл навантаження між екземплярами серверів.

Управління серверними екземплярами здійснює оркестратор, реалізований мовою Python. Оркестратор відстежує стан запущених контейнерів через Docker API та динамічно регулює їх кількість залежно від навантаження на систему. Кожен із серверних екземплярів функціонує в ізольованому Docker-контейнері, що забезпечує стабільність та незалежність їх роботи. Сервіси конвертації файлів реалізовано у межах одного програмного модуля, який містить логіку обробки форматів файлів.

Користувач обирає сервіс конвертації, завантажує файл із диска та отримує конвертований файл. Адміністратор спостерігає за навантаженням на контейнери через відповідні дашборди адмін-панелі, а k6-скрипт генерує навантаження, надсилаючи запити від багатьох віртуальних користувачів (рис. 3.1).

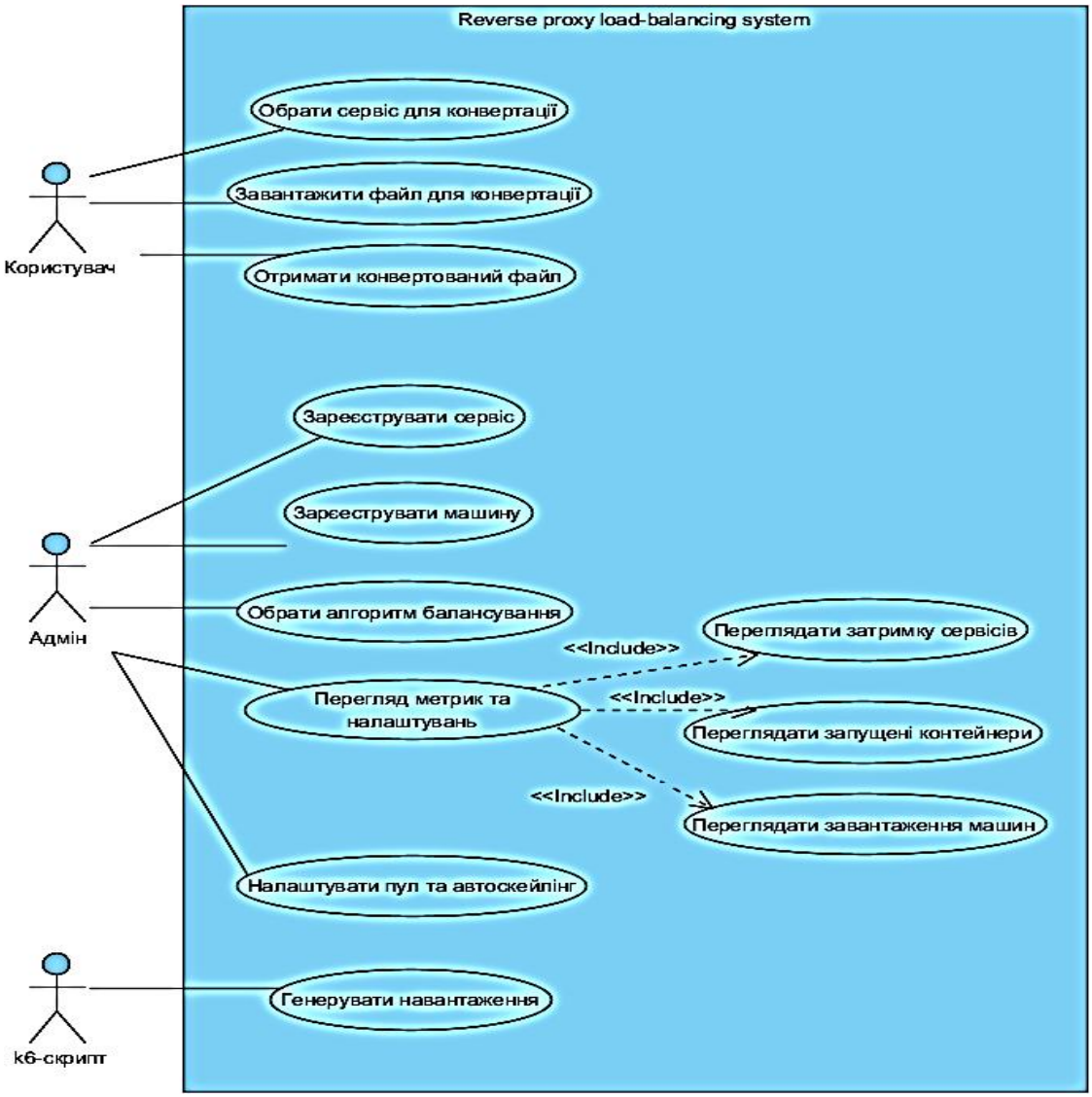


Рисунок 3.1 – Діаграма використання

Для зберігання службових даних системи використовується база даних SQLite. До службових даних належить інформація про сервіси, контейнери, пули та історя запитів.

3.1. Структура проєкту

Проєкт складається з трьох основних директорій. Кожна директорія відповідає за окрему частину системи.

Директорія backend/ містить серверну частину системи. Файл main.py виконує балансування навантаження. Файл admin.py відповідає за адміністрування системи. Скрипт scriptA.sh – це bash-скрипт, який автоматично масштабує Docker-контейнери і керує їх життєвим циклом від моменту активації до зупинки. Файл db_schema.py містить схеми баз даних. Також у цій директорії знаходяться конфігураційні та допоміжні файли.

Директорія frontend/ містить React-застосунок, розбитий на два головних компоненти користувацького та адміністративного інтерфейсів. Основні файли включають src/App.js для сторінки користувача та src/AdminPanel.js для адміністратора. Крім того, тут знаходяться Dockerfile для контейнеризації вебплатформи та nginx.conf для налаштування NGINX як сервера.

Директорія converter/ відповідає за сервіси конвертації файлів і містить модуль app.py, який реалізує логіку конвертації форматів файлів, а також власний Dockerfile для контейнеризації цих сервісів.

Така структура дозволяє чітко розділити компоненти системи. Це спрощує розгортання, масштабування та підтримку проєкту.

3.2. Планувальник життєвого циклу контейнерів

У цьому розділі розміщується опис логіки моніторингу та масштабування, конфігурації та допоміжних функцій планувальника контейнерів. Планувальник контейнерів – це bash-скрипт динамічного керування життєвим циклом Docker-

контейнерів. Він керує запуском, моніторингом та зупинкою контейнерів залежно від навантаження на систему. Лістинг скрипта розміщено у додатку А.

Скрипт реалізує просту форму горизонтального автомасштабування, запускаючи нові контейнери лише тоді, коли поточний контейнер перевантажений, і зупиняє всі контейнери, коли система довго не використовується. Основний процес виконання починається з того, що скрипт видаляє попередній контейнер srv1, якщо той існує, та запускає його знову. Після цього починається головний цикл моніторингу:

— моніториться стан контейнера srv1, і якщо він перевантажений – запускається srv2;

— моніториться srv2, і при його перевантаженні запускається srv3;

— аналогічно для srv3 та srv4.

Контейнер srv4 є останнім у ланцюхі: при його перевантаженні нові контейнери не запускаються.

На початку скрипт визначає основні змінні:

— IMAGE_NAME містить назву Docker-образу, який використовується для запуску контейнерів;

— NETWORK_NAME задає назву мережі Docker-контейнерів;

— MAX_BUSY_COUNT – кількість хвилин, протягом яких контейнер вважається завантаженим перед запуском нового;

— MAX_IDLE_COUNT – кількість хвилин простою, після яких система зупиняє всі контейнери.

Також визначено асоціативний масив PORT, який зв’язує кожен контейнер (srv1-srv4) з відповідним портом на хості (8081-8084).

Скрипт містить кілька допоміжних функцій для роботи з контейнерами:

— container_running перевіряє, чи є контейнер у списку активних процесів;

— container_exists перевіряє, чи існує контейнер взагалі;

— remove_container видаляє контейнер, якщо він існує;

— launch_container запускає новий контейнер, створюючи Docker-мережу (якщо вона ще не існує);

— get_cpu_usage зчитує поточне використання процесора конкретного контейнера за допомогою команди docker stats.

Головна функція скрипту – monitor_until_scaleout_or_idle. Вона приймає два аргументи: ім’я поточного контейнера та ім’я наступного контейнера, який потрібно запустити при перевантаженні.

Функція monitor_until_scaleout_or_idle працює у нескінченному циклі і кожну хвилину виконує такі кроки:

— перевіряє, чи контейнер ще працює;

— зчитує поточне використання процесора;

— якщо навантаження на процесор (CPU) перевищує 30% - збільшує лічильник завантаженості (busy_count) та скидає лічильник простою;

— якщо навантаження на CPU не перевищує 30% - збільшує лічильник простою (idle_count) та скидає лічильник завантаженості.

Коли idle_count досягає значення MAX_IDLE_COUNT, яке складає 6 хвилин, скрипт викликає функцію shutdown_all і завершує свою роботу. Коли busy_count досягає значення MAX_BUSY_COUNT (2 хвилини), скрипт запускає наступний контейнер зі списку. Якщо запущено вже всі чотири контейнери, скрипт виводить повідомлення про досягнення максимальної кількості.

Зупинка системи та видалення всіх контейнерів зі списку PORT відбувається за допомогою функції shutdown_all. Вона перебирає всі імена контейнерів і для кожного активного виконує команди docker stop та docker rm.

Для конкретного завершення при отриманні системного сигналу SIGINT або SIGTERM при натисканні комбінації клавіш Ctrl+C скрипт використовує механізм trap. При отриманні сигналу викликається функція cleanup_on_interrupt, яка запускає shutdown_all і завершує скрипт.

3.3. Розроблений k6-скрипт

Тестові сценарії у k6 пишуться мовою JavaScript, що робить інструмент доступним для розробників без специфічних знань у галузі тестування.

Навантаження у k6 моделюється двома основними способами: через віртуальних користувачів, які імітують одночасну роботу реальних людей, або через задану кількість запитів на секунду, що відтворює реальну пропускну здатність системи.

У межах розробленого програмного продукту k6-скрипт автоматизує збір ключових метрик продуктивності балансувальника навантаження (рис. 2.2).

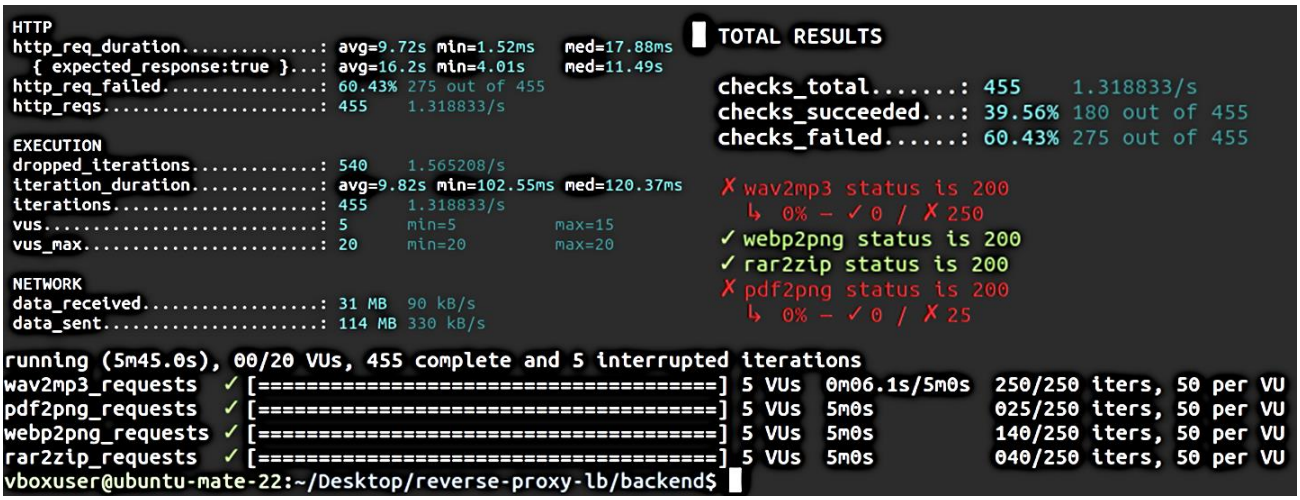


Рисунок 3.2 – Збір ключових метрик навантажувальним тестуванням за допомогою k6

Зокрема відстежуються:

— латентність запитів – час від надсилання запиту до отримання відповіді;

— пропускна здатність – кількість успішно оброблених запитів за одиницю часу;

— частка помилкових відповідей від загальної кількості запитів.

Для перевірки коректності відповідей сервера використовуються перевірки Checks, які валідують умови під час виконання тесту. Для визначення критеріїв проходження або провалу тесту використовуються порогові значення Thresholds, які дозволяють кодифікувати угоди про рівень обслуговування безпосередньо у скрипті.

Завдяки цьому k6 дозволяє не лише зібрати числові показники, а й автоматично визначити, чи відповідає система заданим вимогам продуктивності при різних алгоритмах балансування. Порівнюючи результати тестів для кожного алгоритму, можна об’єктивно визначити більш ефективну стратегію розподілу навантаження для конкретного типу сервісу.

Інструмент навантажувального тестування k6 оптимізований для мінімального споживання ресурсів і розрахований на виконання тестів із високим навантаженням – зокрема спайк-тестів, стрес-тестів і тривалих soak-тестів. Завдяки цьому навіть на комп’ютері із середнім залізом можна змоделювати роботу сотень одночасних користувачів без значного впливу на результати вимірювань [17].

Навантажувальний k6-скрипт, представлений у додатку А, демонструє використання k6 для навантажувального тестування декількох сервісів у межах розробленої програмної системи балансування навантаження. Він імітує одночасну роботу великої кількості користувачів, надсилаючи їм запити до різних API-ендпойнтів, таких як:

— конвертація WAV у MP3;

— конвертація PDF у PNG;

— конвертація WEBP у PNG;

— конвертація RAR у ZIP.

Для кожного сценарію передбачено певну кількість віртуальних користувачів (VUs) та ітерацій, що дозволяє змоделювати реальні умови навантаження.

У скрипті використовується змінна середовища LB_ALG, яка дозволяє під час запуску вибирати алгоритм балансування навантаження для тестування. Це

забезпечує можливість перевіряти продуктивність системи при різних стратегіях розподілу запитів без внесення змін у сам код тесту. За замовчуванням, якщо не було задано жодного алгоритму балансування для змінної середовища, використовується навмання обраний алгоритм із доступного списку.

Кожен сценарій виконує POST-запити, тобто з передачею файлу відповідного формату та обраного алгоритму балансування. Після відправки запиту проводиться перевірка статусу відповіді, що гарантує коректність обробки на сервері.

Коротка пауза «sleep» між запитами імітує природну затримку користувачів та допомагає запобігати штучному «сплеску» навантаження.

Використання k6 у поєднанні з автоматизованим збором метрик має низку переваг:

— такий підхід дозволяє не лише оцінити середню затримку та пропускну здатність сервісів, але й виявити пікові навантаження та потенційні вузькі місця системи;

— можливість порівнювати результати різних алгоритмів дає змогу об’єктивно вибрати найбільш ефективну стратегію для конкретного типу сервісу;

— можливість інтеграції k6 з CI/CD-процесами дозволяє проводити навантажувальні тести автоматично при кожному релізі або зміні конфігурації системи;

— постійний контроль над продуктивністю та стабільністю програмного продукту.

Завдяки високій ефективності та оптимізації k6, навіть при великій кількості віртуальних користувачів інструмент споживає мінімальні ресурси.

Таким чином, використання k6 у межах розробленої системи дозволяє:

— отримати повну картину продуктивності балансувальника навантаження;

— оцінити його стійкість до пікових навантажень;

— ґрунтовно обрати оптимальний алгоритм для забезпечення якісного обслуговування кінцевих користувачів.

Скрипт є гнучким, модульним та легко масштабованим, що робить його важливим інструментом у процесі тестування та подальшого вдосконалення програмного продукту.

3.4. База даних

У цьому підрозділі пояснюється, як організована база даних системи балансування навантаження. Показано, які таблиці існують, як вони пов’язані між собою і яка інформація в них зберігається. Центральне місце займає таблиця Requests, де зберігаються дані про запити користувачів, дані про час обробки, сервіс, інстанс та обраний алгоритм балансування.

Також розглядаються таблиці сервісів, контейнерів, машин, пулів та правил автомасштабування. Завдяки цим таблицям система може відстежувати, який сервер обробляє запити, наскільки він завантажений і чи потрібне масштабування.

Розділ показує, що база даних побудована логічно і без повторюваних даних, а зв’язки між таблицями дозволяють швидко знаходити потрібну інформацію. База даних не лише зберігає інформацію, а й допомагає контролювати роботу всієї системи.

3.4.1. Архітектура бази даних

На рис. 3.11 наведена концептуальна модель бази даних, яка відображає взаємозв'язки між таблицями, їх первинні та зовнішні ключі.

Таблиця «Requests» є ключовою – у ній зберігаються дані щодо пропускної здатності та завантаженості екземплярів серверів, а також обрана стратегія балансування.

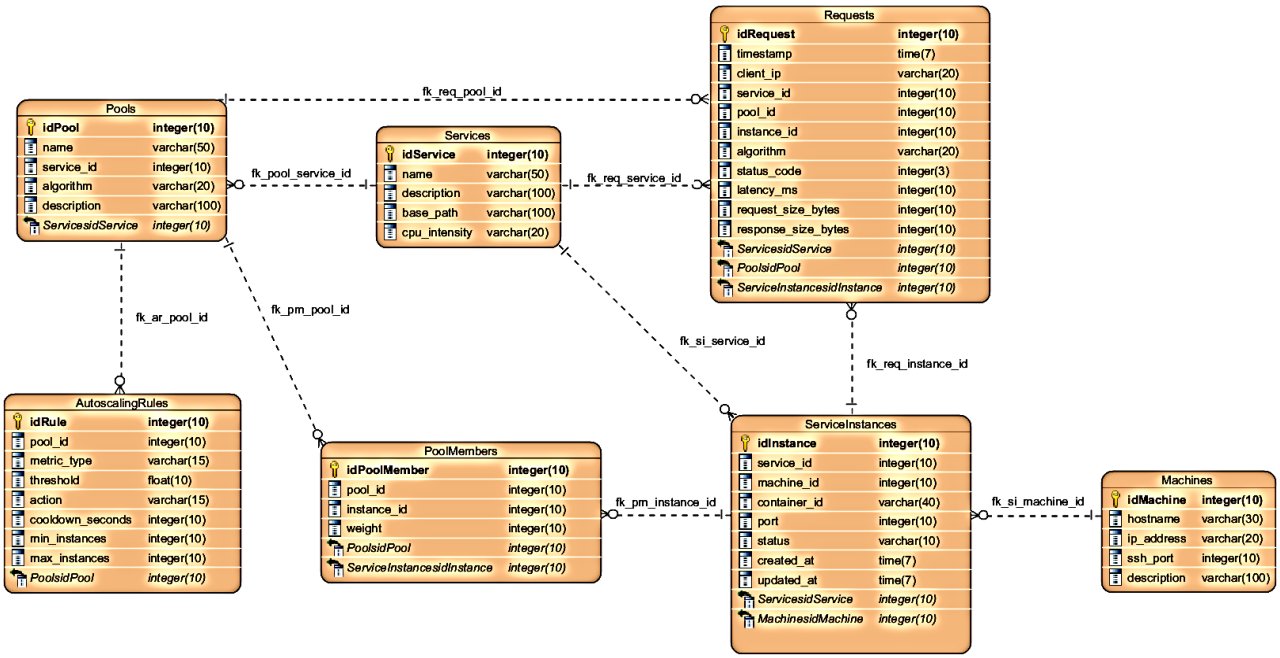


Рисунок 3.3 – Концептуальна модель бази даних

База даних приведена до третьої нормальної форми (3НФ), згідно з вимогами якої:

— стовпці і рядки містять атомарні (неподільні) значення атрибутів і не містять повторюваних груп;

— кожен описовий атрибут функціонально повно залежить від первинного ключа;

— відношення взаємно незалежні і повністю залежать лише від первинного ключа (транзитивність) [18].

Структура бази даних побудована таким чином, щоб забезпечити зручне зберігання наступної інформації:

— сервіси конвертації;

— екземпляри сервісів;

— запущені контейнери;

— результати роботи балансування.

Центральне місце в моделі займає таблиця «Requests», яка пов’язує між собою сервіси, пули та конкретні інстанси, що обробляли запити користувачів. Завдяки цьому забезпечується можливість простежити повний шлях проходження кожного запиту через систему.

3.4.2. Опис таблиць бази даних

Таблиця «Requests» – центральна таблиця логування, що фіксує кожен HTTP-запит, який пройшов через балансувальник. Аналіз зібраних даних у цій таблиці дозволяє адміністратору програмної системи відповісти на питання про те, чи були звернення та від кого, до якого сервісу, який інстанс обробив запит та скільки часу це зайняло.

Таблиця є основним джерелом даних для аналізу продуктивності та порівняння алгоритмів балансування. Вона містить наступні поля:

— idRequest – первинний ключ;

— timestamp – час надходження запиту;

— client_ip – IP-адреса клієнта;

— service_id –ідентифікатор сервіса конвертації;

— pool_id – ідентифікатор пулу, що обробив запит;

— instance_id – ідентифікатор конкретного інстанса-виконавця;

— algorithm – обраний алгоритм балансування;

— status_code – HTTP-код відповіді;

— latency_ms – час виконання запиту, [мс];

— request_size_bytes – розмір вхідного файлу в байтах;

— response_size_bytes – розмір вихідного файлу в байтах.

Поля ServiceidService, PoolsidPool та ServiceInstancesidInstance є зовнішніми ключами, які забезпечують зв’язок таблиці «Requests» із таблицями Services, Pools та ServiceInstances. Використання зовнішніх ключів гарантує, що кожне значення зовнішнього ключа відповідає існуючому запису в пов’язаній таблиці, забезпечуючи узгодженість даних та коректність зв’язків між сутностями бази даних [19].

Окрім забезпечення цілісності даних, зовнішні ключі значно спрощують виконання аналітичних запитів до бази даних. Накопичені в таблиці «Requests» дані можуть використовуватися для:

— побудови статистики щодо популярності сервісів;

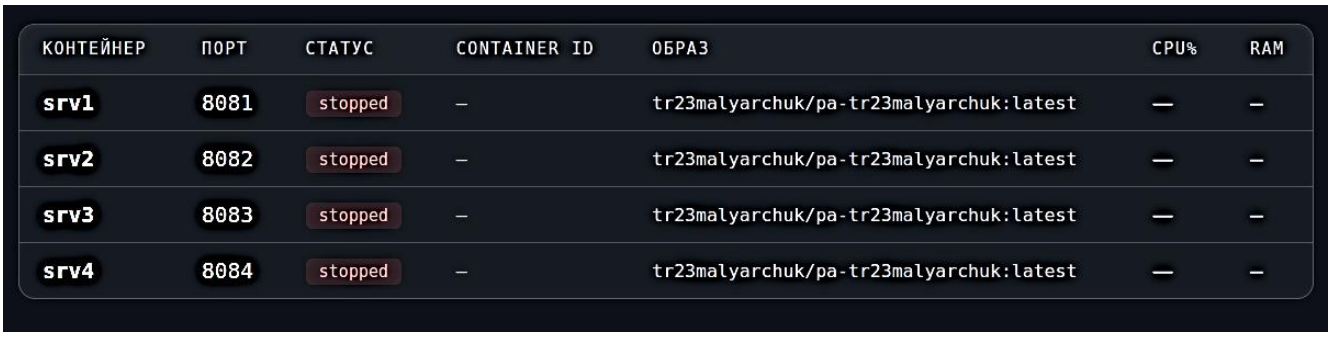
— визначення рівня завантаженості окремих інстансів та ефективності різних алгоритмів балансування;

— виявлення вузьких місць системи та оцінки якості її роботи за різних умов навантаження, що забезпечується аналізом значень полів latency_ms, status_code та algorithm.

Таким чином, таблиця «Requests» виконує не лише функцію журналу подій, а й слугує основою для моніторингу, діагностики та подальшої оптимізації програмної системи.

Таблиця «ServiceInstances» зберігає інформацію про окремі екземпляри сервісів, які запущені у вигляді Docker-контейнерів на різних портах і машинах.

Кожен запис у цій таблиці відповідає одному контейнеру. Наприклад, якщо образ запущений одночасно на чотирьох інстансах – у таблиці буде чотири окремі записи, кожен зі своїм портом, контейнером і статусом (рис. 3.4).



КОНТЕЙНЕР	ПОРТ	СТАТУС	CONTAINER ID	ОБРАЗ	CPU%	RAM
srv1	8081	stopped	–	tr23malyarchuk/pa-tr23malyarchuk:latest	—	–
srv2	8082	stopped	–	tr23malyarchuk/pa-tr23malyarchuk:latest	—	–
srv3	8083	stopped	–	tr23malyarchuk/pa-tr23malyarchuk:latest	—	–
srv4	8084	stopped	–	tr23malyarchuk/pa-tr23malyarchuk:latest	—	–

Рисунок 3.4 – Інформація про екземпляри сервісів

Таблиця пов’язана з двома іншими: «Services» – щоб знати, який саме сервіс виконує контейнер, і «Machines» – щоб знати, на якій машині він працює.

Балансувальник використовує цю таблицю щоб зрозуміти, куди саме направляти запит користувача – на яку IP-адресу і порт. Якщо контейнер зупинився, його статус змінюється на stopped, і балансувальник більше не направляє туди запити.

Поля created_at і updated_at дозволяють відслідковувати коли контейнер був запущений і коли востаннє змінювався його стан – це корисно для моніторингу та діагностики проблем.

Таблиця «Machines» зберігає інформацію про фізичні або віртуальні комп’ютери, на яких можуть запускатися контейнери з сервісами конвертації.

Кожен запис – це окрема машина в мережі. Адміністратор додає машини через вебінтерфейс, щоб система знала, які сервери доступні для роботи. Якщо якась машина виходить з ладу, її запис можна видалити, і система більше не буде намагатися запускати на ній контейнери.

Таблиця має наступні поля:

— idMachine – унікальний ідентифікатор машини. Використовується в якості зовнішнього ключа в ServiceInstances;

— hostname, ip_address – дозволяють системі знайти машину в мережі;

— ssh_port – використовується для підключення до неї з метою керування контейнерами;

— description – необов’язкове поле і слугує виключно для зручності адміністратора, а саме для тих випадків, коли виникає бажання вказати розташування сервера або його роль у системі.

Наявність коректних даних у Machines забезпечує надійність розгортання контейнерів і можливість масштабування; помилкова IP-адреса робить неможливим доставку запитів до контейнера, що обслуговується на цьому віртуальному хості.

У таблиці «Services» зберігається перелік усіх типів конвертації файлів, які підтримує система. Кожен запис описує бізнес-логіку сервісу, його відношення до пулів та інстансів.

Структура полів:

— idService – унікальний ідентифікатор сервісу;

— name – коротка назва сервісу;

— description – докладний опис призначення сервісу;

— base_path – визначає URL-адресу, за якою клієнт звертається до цього сервісу через балансувальник. Завдяки цьому системі відомо, який запит до якого сервісу відноситься);

— cpu_intensity – відображає пріоритетність типу конвертації для процесора. Наприклад, обробка PDF-файлів вимагає більше ресурсів, ніж конвертація зображень. Дозволяє підбирати більш ресурсоємкі машини або коригувати правила автомасштабування.

Наявні зв’язки 1:M (one-to-many):

— один «Service» – багато «Pools»;

— один «Service» – багато «ServiceInstances»;

— один «Service» – багато «Requests».

Налаштування cpu_intensity дозволяє правильно розподіляти ресурси, а помилкова маршрутизація base_path призведе до некоректної обробки запитів.

Виділення сервісів в окрему сутність дозволяє централізовано керувати параметрами різних типів конвертації. При додаванні нового сервісу немає необхідності змінювати структуру бази даних або логіку балансування навантаження. Достатньо створити новий запис та прив’язати до нього необхідні пули й інстанси.

Таблиця «PoolMembers» описує відношення між конкретними інстансами сервісів та пулами, у які вони входять. Це сполучна таблиця із типом зв’язків many-to-many між «Pools» та «ServiceInstances».

Структура полів:

— idPoolMember – унікальний ідентифікатор зв’язку;

— pool_id – пул, до якого належить член;

— instance_id – інстанс сервісу;

— weight – дозволяє задати пріоритет окремого контейнера – чим більше значення, тим більше запитів він отримує.

Налаштування weight дозволяє тонко керувати розподілом навантаження. Неправильні ваги можуть перевантажити окремі інстанси.

Таблиця «Pools» представляє собою логічні групи контейнерів (пули балансування), між якими розподіляються вхідні запити користувачів. Кожен пул прив’язаний до одного сервісу і використовує власний визначений алгоритм балансування.

Адміністратор може створювати окремі пули для різних сервісів і налаштовувати стратегію розподілу навантаження незалежно для кожного з них. Таблиця має наступну структуру полів:

— idPool, name – унікальний ідентифікатор пулу та його назва;

— service_id – сервіс, до якого прив’язаний пул;

— algorithm – обраний алгоритм балансування;

— description – детальний опис, додаткові примітки, тощо;

— ServicesidService – зовнішній ключ.

Зв’язки у таблиці:

— один «Pool» – багато «PoolMembers»;

— один «Pool» – багато «AutoscalingRules»;

— один «Pool» – багато «Requests».

Саме на рівні пулів реалізується логіка балансування навантаження. Такий підхід дозволяє використовувати різні алгоритми для різних сервісів одночасно. Наприклад, для ресурсоємних операцій може застосовуватися алгоритм Least Connections, тоді як для менш вимогливих сервісів достатньо Round Robin.

Таблиця «AutoscalingRules» зберігає умови, за якими система автоматично запускає або зупиняє контейнери.

Кожне правило відслідковує певну метрику – наприклад, завантаженість CPU – і при досягненні порогового значення виконує відповідну дію. Наступні поля таблиці значним чином впливають на працездатність екосистеми контейнерів:

— idRule – ідентифікатор правила;

— pool_id – пул, для якого застосовується правило;

— metric_type – тип метрики;

— threshold – порогове значення метрики, при якому спрацьовує правило;

— action – дія (масштабуватись новим контейнером чи зупинити);

— cooldown_seconds – період, протягом якого заборонено повторне застосування правила (не дозволяє системі масштабуватися занадто часто);

— min_instances, max_instances – обмежують допустиму кількість контейнерів у пулі.

Правильно налаштовані правила запобігають недовантаженню або перевантаженню екосистеми контейнерів, забезпечуючи відновлюваність під час високого навантаження.

Неправильні пороги або короткий cooldown можуть спричинити нестабільність у системі.

Наявність окремої таблиці правил масштабування робить систему більш адаптивною до змін навантаження. Усі параметри масштабування можуть змінюватися адміністратором без необхідності перезапуску сервісів або внесення змін до коду програмного продукту, що є спрощенням громіздкого процесу налаштування системи. Це дозволяє швидко адаптувати систему до нових умов експлуатації.

3.5. Сервіси конвертації

Розроблена вебплатформа надає користувачам чотири незалежних сервіси конвертації файлів різних форматів. Кожен сервіс реалізований як окремий ендпойнт у межах одного FastAPI-застосунку, запакованого в Docker-образ. Завдяки цьому можна запускати кілька екземплярів одного й того самого образу одночасно та рівномірно розподіляти між ними запити від користувачів за допомогою балансувальника навантаження. Сервіси суттєво відрізняються між собою за обчислювальними вимогами: конвертація PDF-документів є найважчою операцією для процесора, тоді як конвертація зображень виконується значно швидше. Ця різниця у навантаженні робить платформу хорошим прикладом гетерогенного робочого навантаження, на якому можна наочно порівняти ефективність різних алгоритмів балансування. Кожен сервіс приймає файл через HTTP POST-запит у вигляді multipart/form-data і повертає результат конвертації у відповідь.

У разі помилки балансувальник автоматично повторює запит на іншому сервері – до трьох осіб.

Неоднорідність обчислювальних вимог сервісів конвертації суттєво впливає на ефективність статичних стратегій балансування, що обумовлює доцільність застосування адаптивних алгоритмів в умовах гетерогенного навантаження [20].

3.5.1. PDF2PNG

Сервіс PDF2PNG призначений для конвертації PDF-документів у набір PNG-зображень – по одному зображенню на кожну сторінку документа. Результат повертається користувачу у вигляді ZIP-архіву з отриманими зображеннями.

Реалізація сервісу побудована на бібліотеці pdf2image, яка використовує утиліту Poppler для растеризації PDF-сторінок. Кожна сторінка документа растеризується з роздільною здатністю 150 DPI і зберігається як окреме PNG-зображення у буфер пам’яті. Після обробки всіх сторінок зображення пакуються в ZIP-архів за допомогою вбудованого модуля zipfile із стисненням ZIP_DEFLATED.

З точки зору навантаження на процесор цей сервіс є найважчим серед усіх чотирьох. Час обробки залежить від кількості сторінок у документі та їхньої складності. Для великих документів може знадобитися значний обсяг оперативної пам’яті та більше часу на пакетну обробку. Сервіс оптимізований для одночасної роботи з кількома запитами за допомогою контейнеризації, а також підтримує масштабування через додавання нових екземплярів сервісу і пулі балансувальника навантаження.

3.5.2. WAV2MP3

Сервіс WAV2MP3 виконує конвертацію аудіофайлів формату WAV у стиснений формат MP3. Формат WAV є нестисненим і займає значно більше місця на диску порівняно з MP3, тому конвертація корисна для зменшення розміру аудіофайлу без суттєвої втрати якості звучання.

Реалізація сервісу побудована на виклику зовнішньої утиліти ffmpeg через стандартний Python-модуль із назвою subprocess. Вхідний WAV-файл зберігається у тимчасовий файл на диску, після чого ffmpeg виконує конвертацію і записує результат у новий тимчасовий MP3-файл.

Після завершення конвертації файл зчитується у пам’ять і повертається у відповіді, а тимчасові файли видаляються незалежно від результату операції – у блоці finally. За рівнем навантаження на процесор цей сервіс займає середню позицію. Час обробки залежить від тривалості аудіофайлу, як і у випадку з PDF2PNG-сервісом. Коротке повідомлення конвертується за мілісекунди, тоді як довгий аудіозапис може займати кілька секунд. Інструмент ffmpeg є добре оптимізованим під такі навантаження, тому конвертація відбувається ефективно навіть на обладнанні середнього класу.

3.5.3. WEBP2PNG

Сервіс WEBP2PNG виконує конвертацію зображень формату WEBP у формат PNG. Формат WEBP було розроблено компанією Google. Для WEBP формату характерний кращий ступінь стиснення порівняно з PNG, проте деякі застосунки та сервіси досі не підтримують його відтворення. Конвертація у PNG вирішує цю проблему сумісності.

Реалізація побудована на бібліотеці Pillow – одній з найбільш поширених Python-бібліотек для роботи із зображеннями. Вхідний WEBP-файл завантажується у пам’ять у відкривається за допомогою Image.open(). Перед збереженням перевіряється кольоровий режим зображення: якщо він відрізняється від RGB або RGBA, зображення автоматично конвертується у режим RGBA, що забезпечує коректне збереження прозорості. Результат записується у буфер пам’яті у форматі PNG і повертається у відповіді.

3.5.4. RAR2ZIP

Сервіс RAR2ZIP виконує перепакування архівів формату RAR у формат ZIP. Формат RAR є пропрієтарним і для його розпакування потрібне спеціальне програмне забезпечення, тоді як ZIP підтримується вбудованими засобами більшості операційних систем без встановлення додаткових програм.

Особливістю цього сервісу є непередбачуваність часу обробки. На відміну від PDF або аудіо, де час конвертації залежить від розміру файлу, для RAR-архіву він також залежить від кількості файлів всередині та їхньої структури вкладеності. Архів, що містить тисячі дрібних файлів, обробляється довше, ніж архів такого ж розміру з одним великим файлом.

Для роботи з архівами використовується бібліотека rarfile, яка забезпечує доступ до вмісту RAR-файлів через програмний інтерфейс Python. Після розпакування вміст архіву повторно упаковується у формат ZIP із збереженням структури каталогів та файлів. Відповідно, користувач отримує результат у більш поширеному форматі, який підтримується більшістю сучасних операційних систем і програм для роботи з архівами.

4 РОБОТА КОРИСТУВАЧІВ З ПРОГРАМНОЮ СИСТЕМОЮ

У цьому розділі описано, як користувачі працюють із програмною системою. Розділ складається з двох частин. Перша частина описує роботу зі сторінкою користувача: вибір сервісу конвертації, завантаження файлу та отримання результату. Друга частина описує роботу з панеллю адміністратора: реєстрацію сервісів і машин, налаштування балансування навантаження та моніторинг стану системи. Для кожного кроку наведено відповідні ілюстрації інтерфейсу. Це дозволяє наочно побачити, як система працює на практиці.

4.1. Робота зі сторінкою користувача

Сторінка користувача – вебсторінка на основі React-фреймворку, що дозволяє взаємодіяти з сервісами конвертації через різнокольорові функціональні плитки. Кожна плитка відповідає окремому сервісу конвертації (рис. 4.1).

Після вибору необхідного сервісу користувач може завантажити файл для обробки, і система підтримує лише один спосіб для отримання такого файлу – із локального диску. Користувач отримує результат конвертації у відповідному форматі.

Взаємодія із системою здійснюється через будь-який сучасний веббраузер. Інтерфейс розроблено таким чином, щоб забезпечити простий доступ до всіх доступних сервісів та звести до мінімуму кількість дій, необхідних для виконання конвертації файлів.

Рисунок 4.1 – Сторінка користувача

Інтерфейс спроєктовано таким чином, щоб скоротити кількість дій, необхідних для виконання конвертації. Користувачу достатньо було виконати лише три кроки: обрати сервіс конвертації, завантажити файл та натиснути кнопку «Надіслати на конвертацію».

Реалізація сторінки користувача на основі React дозволяє динамічно оновлювати стан інтерфейсу без перезавантаження сторінки. Компонентна архітектура React забезпечує чітке розділення логіки вибору сервісу, завантаження файлу та відображення результату.

Користувач заходить на вебсторінку і обирає сервіс конвертації з доступних. Панель завантаження за замовчуванням неактивна (рис. 4.2).

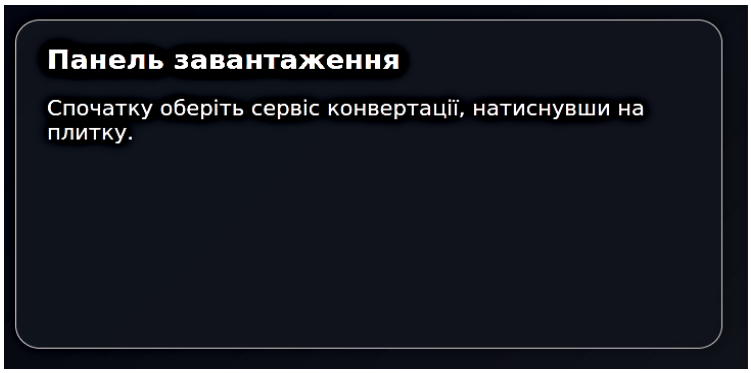


Рисунок 4.2 – Неактивний стан панелі

Як тільки обрано бажаний сервіс і активовано відповідну плитку натисканням ЛКМ, на панелі починає відображатись інструкція до завантаження (рис. 4.3).

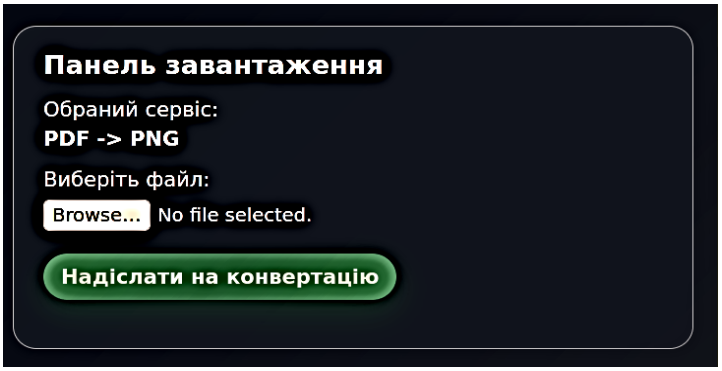


Рисунок 4.3 – Панель завантаження активовано

Щоб здійснити конвертацію, надається можливість вкласти файл із локального диску. Користувач обирає заготовлений для конвертації файл (рис. 4.4).

Рисунок 4.4 – Вибір файла для конвертації

Файл було обрано, про що свідчить оновлений статус панелі завантаження із назвою обраного файла (рис. 4.5).

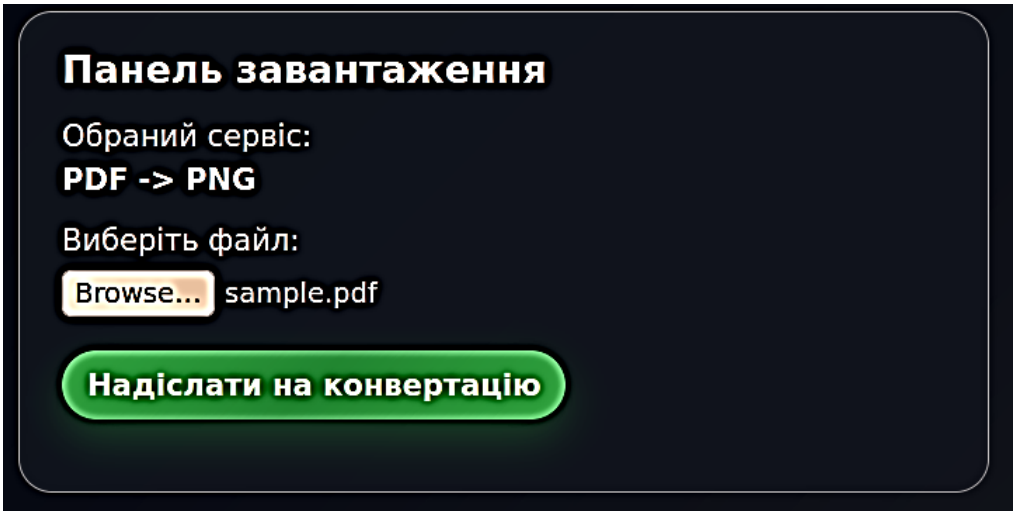


Рисунок 4.5 – Зміна статусу панелі після вкладення файла

Із натисканням на кнопку «Надіслати на конвертацію» здійснюється конвертація, після чого користувач побачить вікно підтвердження, де можна обрати назву конвертованого файла і його розташування (рис. 4.6).

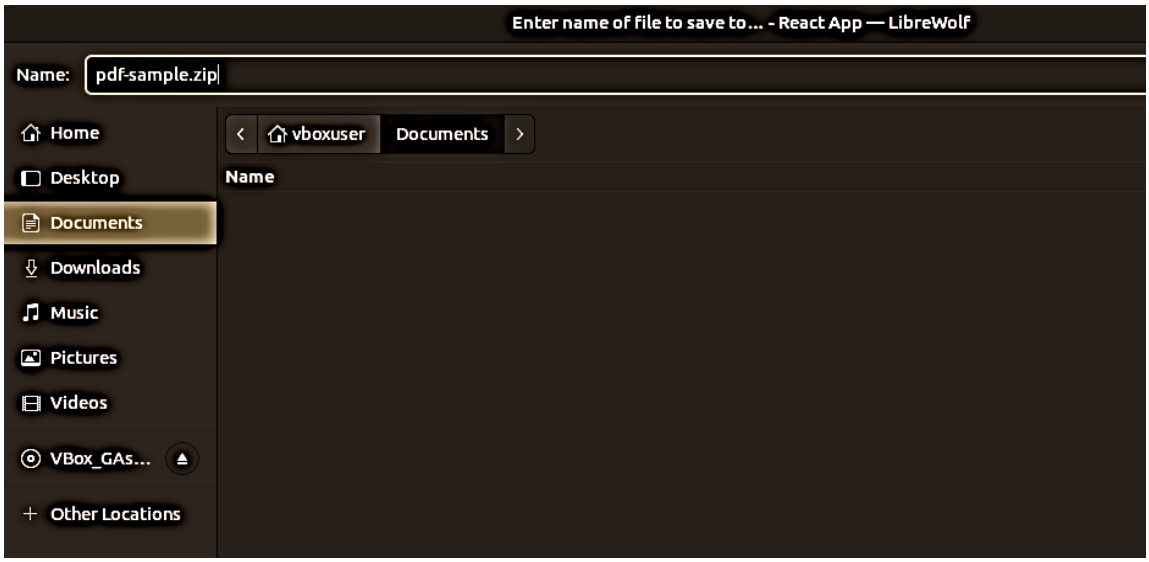


Рисунок 4.6 – Вибір назви і розташування конвертованого файлу

Файл було успішно конвертовано і збережено. Перевірити це можна у завантаженнях поточного веббраузера (рис. 4.7).

Рисунок 4.7 – Завантаження

При поверхневому огляді конвертованого архіву із .png зображеннями видно, що сервіс конвертації спрацював коректно, вклавши зображення кожної pdf-сторінки (рис. 4.8).

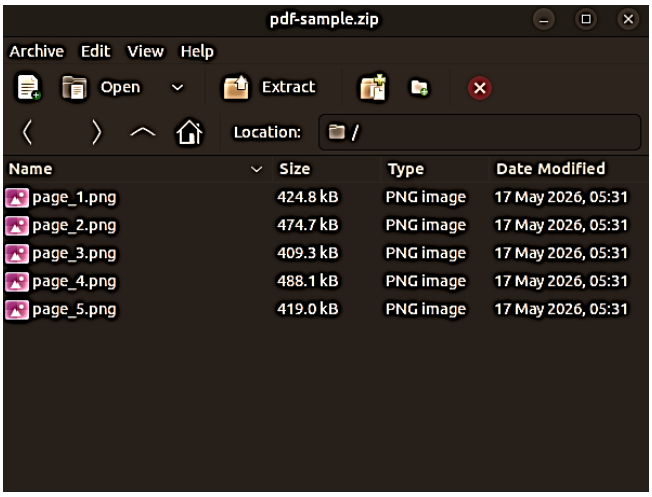


Рисунок 4.8 – Огляд архіву конвертованих pdf-сторінок

Додатково можна відкрити одне із зображень і перевірити його якість (рис. 4.9).

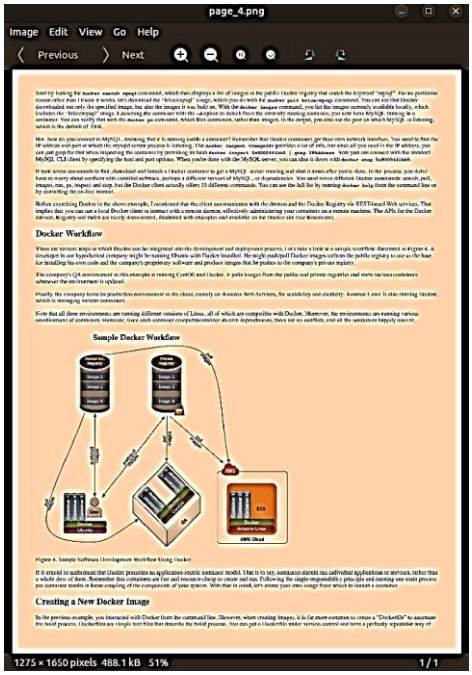


Рисунок 4.9 – Перегляд одного із конвертованих зображень

Аналогічно із іншими сервісами конвертації.

4.2. Робота із панеллю адміністратора

Сторінка адміністратора є сервером конфігурації з інтуїтивним вебінтерфейсом, що дозволяє адміністратору реєструвати нові сервіси та пов’язувати їх із відповідними Docker-образами, реєструвати робочі машини, а також налаштовувати пули машин, доступних для кожного сервісу. Уся метаінформація зберігається у реляційній базі даних SQLite.

Вебінтерфейс також забезпечує можливість моніторингу і візуалізації статистики в режимі реального часу:

— рівень навантаження на сервери і машини;

— стан запущених контейнерів;

— затримку відповіді сервісів.

Сторінка адміністратора є мультифункціональним інструментом моніторингу і конфігурації для системи балансування навантаження на основі зворотного проксі-сервера. Вона дозволяє швидко реагувати на зміни навантаження та своєчасно виконувати необхідні процедури для підтримки стабільної роботи системи.

Завдяки інтеграції з базою даних адміністратор має доступ до детальної інформації про всі активні сервіси та пули контейнерів, що забезпечує ефективне планування ресурсів і прогнозування роботи системи.

Вкладка «Services & Images» представляє собою панель, що надає адміністратору можливість відстежувати, який сервіс спричиняє найбільше навантаження на процесор, поточний образ та наявність оновлень для нього (рис. 4.10).

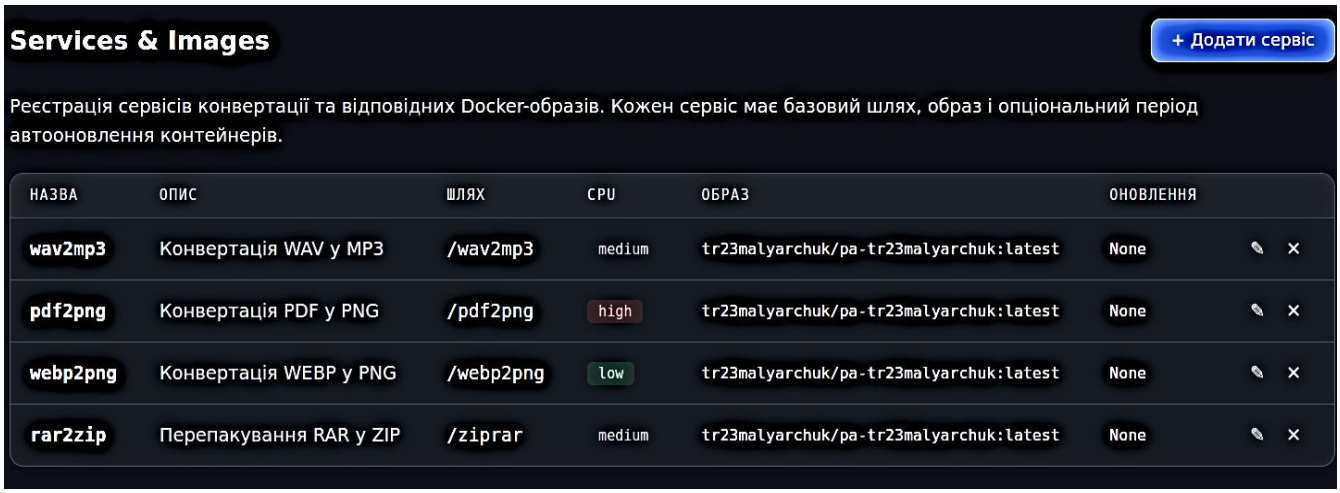


Рисунок 4.10 – Вкладка «Services & Images»

Інформація про зареєстровані сервіси зберігається у базі даних і використовується балансувальником навантаження для маршрутизації вхідних запитів до відповідних контейнерів [21].

Вкладка «Машини» надає адміністратору повну картину пристроїв, що залучені у мережу і між якими відбувається розподіл запитів (рис. 4.11).

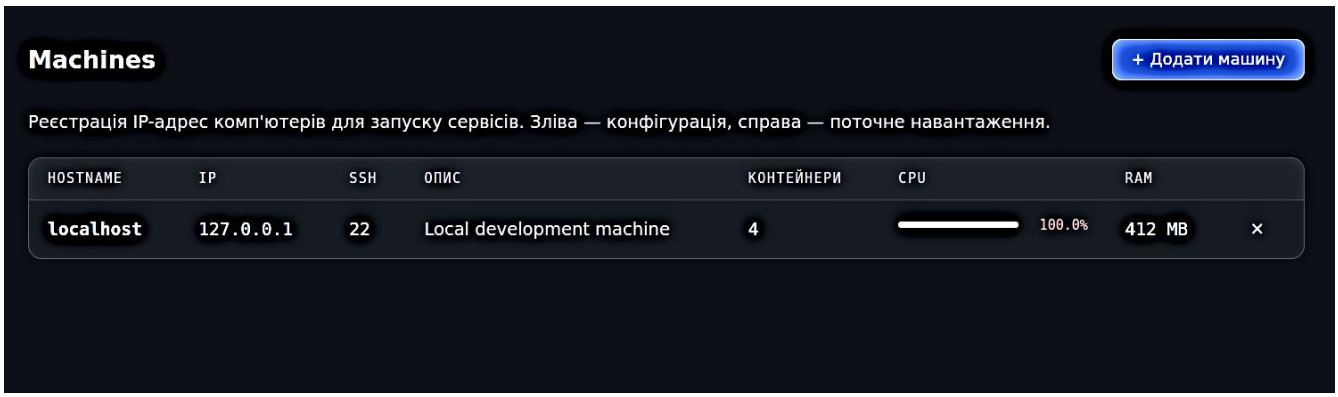


Рисунок 4.11 – Вкладка «Machines»

Для кожної зареєстрованої машини відображається її IP-адреса, порт SSH, опис та поточна кількість запущених контейнерів. Це дозволяє адміністратору оперативно відстежувати стан вузлів мережі та керувати ресурсами.

Вкладка «Стратегії балансування» містить інфографіку по порівнянню між собою алгоритмів балансування по критерію середнього часу відповіді та кількості отриманих запитів (рис. 4.12).

Рисунок 4.12 – Вкладка «Balancing Strategies»

Представлені графіки дають можливість наочно порівняти ефективність реалізованих стратегій балансування.

У вкладці «Завантаженість машин» за інформаційними картками можна моніторити завантаженість центрального процесора (CPU), оперативної пам’яті (RAM) екземплярів серверів (рис. 4.13). Ці дані дозволяють адміністратору оцінювати ефективність розподілу навантаження між машинами в реальному часу. Крім того, вкладка дає змогу швидко виявляти перевантажені або недовантажені сервери та своєчасно приймати рішення щодо масштабування або перенаправлення запитів.

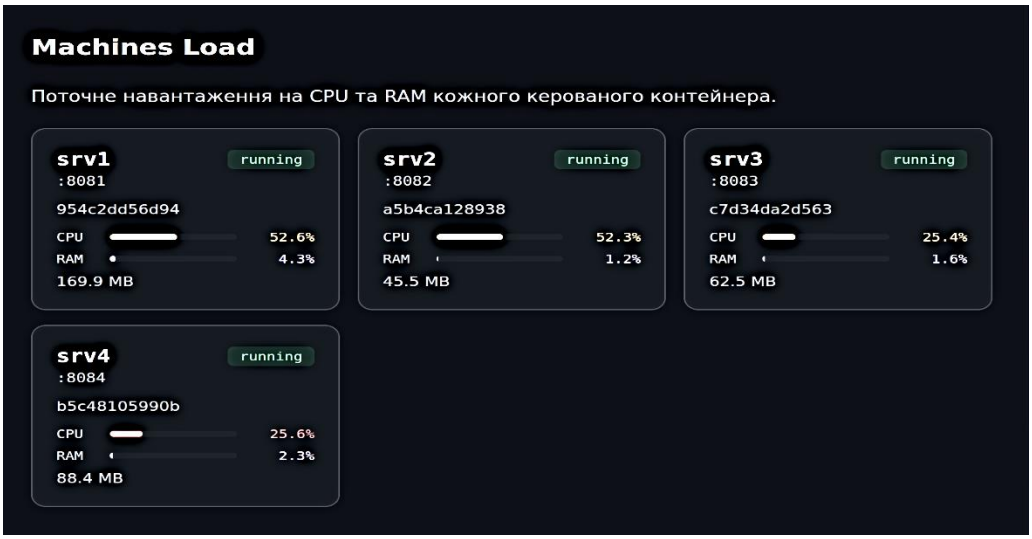


Рисунок 4.13 – Вкладка «Machines Load» – інформаційні картки

Графік окремо у наочному вигляді показує завантаженість контейнерів виходячи із навантаження на центральний процесор (рис. 4.14).

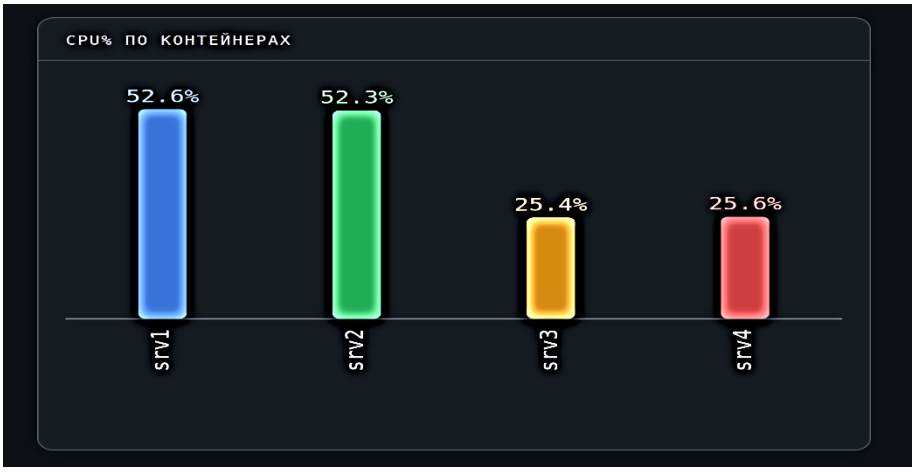


Рисунок 4.14 – Вкладка «Machines Load» – графік

Графічне відображення завантаженості CPU дозволяє відстежувати динаміку навантаження у часі та візуально порівнювати стан окремих контейнерів між собою.

Вкладка «Запущені контейнери» показує назви контейнерів, їх статуси, версії, ідентифікатори, а також завантаженість процесору та оперативної пам’яті (рис. 4.15).

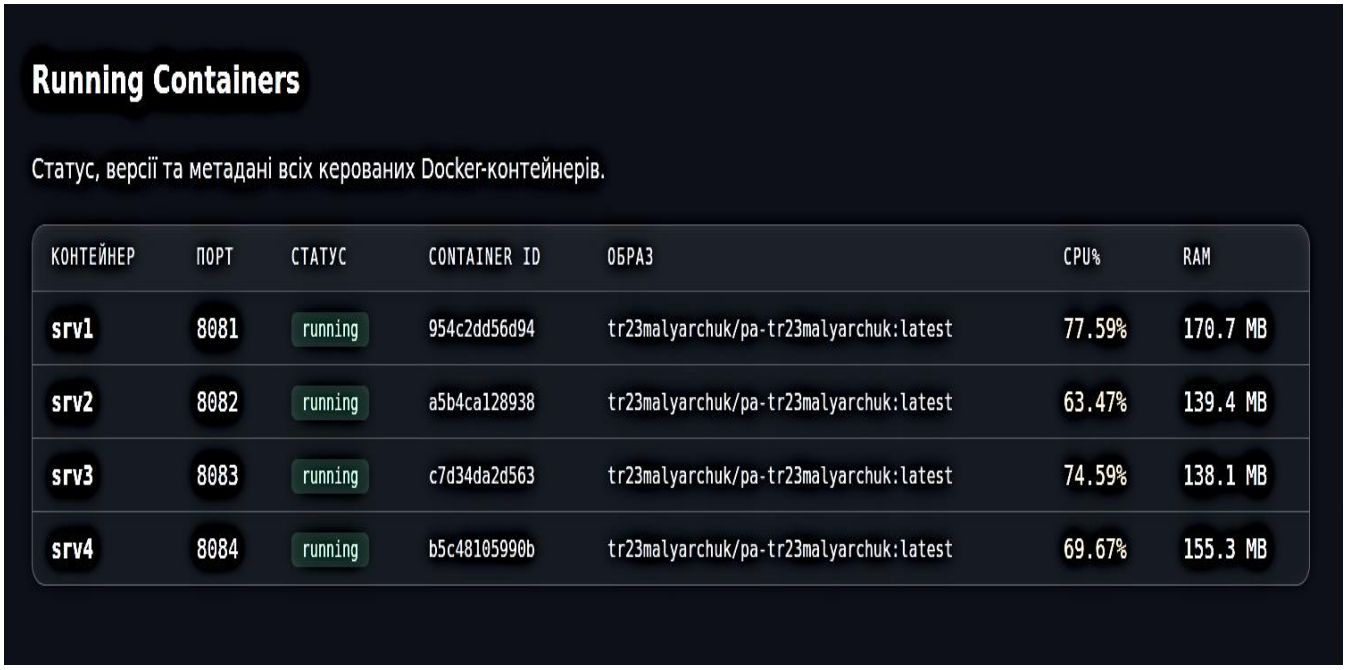


Рисунок 4.15 – Вкладка «Running Containers»

Знаходячись у цьому розділі, адміністратор може порівнювати показники навантаження на CPU та RAM кожного з контейнерів без необхідності звертатися до командного рядка.

Вкладка «Пропускна здатність сервісів» («Service Latency») має за ціль порівняти пропускну здатність серверів і сервісів визначенням середніх показників. Результати цілком наочно відображаються на графіках свічок (рис. 4.16).

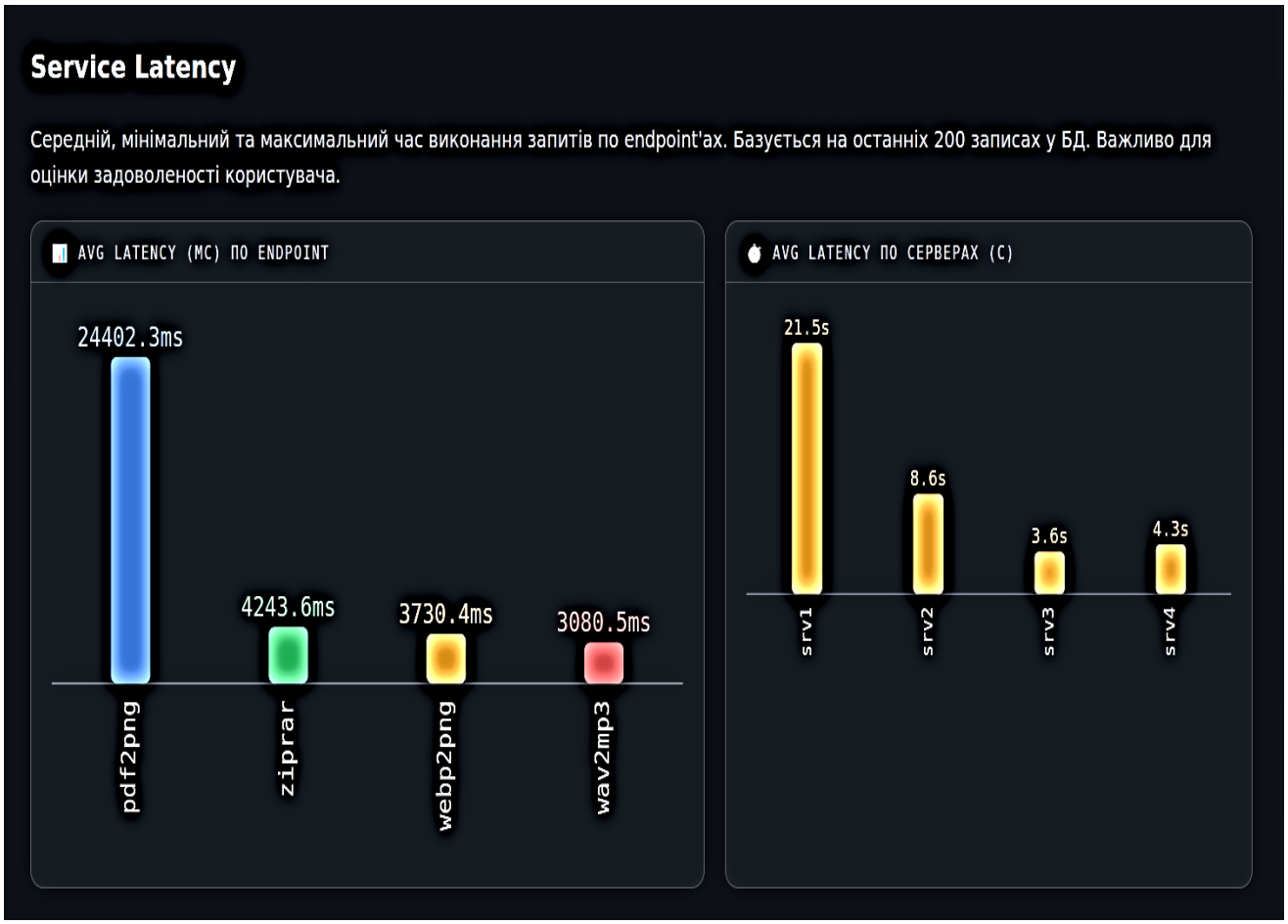


Рисунок 4.16 – Вкладка «Service Latency» – середня пропускна здатність по адресам та серверах

У другій частині дашборду вкладки представлено інформаційні картки зі свічковими графіками, що відображають певну закономірність: PDF-сервіс конвертації більш вимогливий ніж сервіс конвертації аудіо форматів (рис. 4.17). Це дозволяє адміністратору швидко оцінити ресурсоємність різних сервісів і прийняти обґрунтоване рішення щодо оптимізації розподілу навантаження.

Рисунок 4.17 – Вкладка «Service Latency» – інформаційна таблиця та графіки запитів

Наведені дані підтверджують теоретичне припущення про гетерогенність обчислювальних вимог різних сервісів конвертації. PDF-сервіс демонструє значно вищу латентність порівняно з іншими сервісами за рахунок необхідності растеризації кожної сторінки документа в окреме графічне зображення. З цього випливає, що під час планування розподілу навантаження слід враховувати ресурсоємність сервісів, щоб уникнути перевантаження потужних екземплярів. Крім того, такі дані допомагають оцінювати ефективність вибраного алгоритму балансування для кожного типу сервісу.

ВИСНОВКИ

Проведено огляд чотирьох існуючих систем балансування навантаження, показано актуальність розподілу запитів між кількома серверами для підвищення відмовостійкості та масштабованості. На основі аналізу функціональних можливостей, вимог до продуктивності та підтримки адаптивних алгоритмів було обрано NGINX як зворотний проксі-сервер для реалізації балансувальника завдяки його стабільності, гнучкості у налаштуваннях та високої продуктивності у середовищі з великим навантаженням.

Для реалізації програмної системи було обрано програмно-технологічний стек: React для створення вебплатформи; NGINX як балансувальник із зворотним проксі; Docker для контейнеризації сервісів конвертації; SQLite для зберігання метаданих про сервіси, пули, інстанси, правила масштабування; Bash/Python для створення скриптів керування життєвим циклом контейнерів та автомасштабування.

Розроблено та реалізовано програмну систему балансування навантаження вебсервісів, розгорнутих у Docker-контейнерах.

Реалізовано п’ять стратегій балансування навантаження та проведено їх порівняльний аналіз виходячи з швидкості обробки запитів і рівномірності розподілу навантаження. Оцінено ресурсоємність сервісів конвертації і показано, що PDF-конвертація є найбільш обчислювально витратною. Показано ефективність стратегії Power of Two, і водночас з’ясовано, що вибір оптимальної стратегії балансування є контекстно залежним і визначається конкретними вимогами системи – пріоритетом швидкості обслуговування, рівномірністю розподілу ресурсів або необхідністю підтримки сесій користувача.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Malyarchuk B. V., Demchyshyn A. A. Reverse Proxy Load-Balancing System // Сучасні проблеми наукового забезпечення енергетики: безпека, стійкість, IT та екологічний моніторинг : XXIII Міжнар. наук.-практ. конф. мол. вчених і студ., м. Київ, 17-22 квіт. 2026 р. Київ : КПІ ім. Ігоря Сікорського, Вид-во «Політехніка», 2026. P. 419-421. URL: https://iate.kpi.ua/uploads/p_182_80374158.pdf (дата звернення: 02.06.2026).
2. Clements P., Garlan D., Bass L. та ін. Software architecture: introducing IEEE Standard 1471. IEEE Software. 2001. Vol. 18, № 6. URL: https://www.researchgate.net/publication/2955414_Software_architecture_introducing_IEEE_Standard_1471 (дата звернення: 01.06.2026).
3. Abhishek S. Dynamic Load Balancing of Microservices in Kubernetes Clusters using Service Mesh. MSc Cloud Computing. 2022. P. 1–20.
4. Understanding Load Balancing Algorithms and Strategies. OneUptime Blog. URL: https://oneuptime.com/blog/post/2026-02-20-load-balancing-algorithms/view (дата звернення: 01.06.2026).
5. Comparing Load Balancing Algorithms. JSCAPE. URL: https://www.jscape.com/blog/load-balancing-algorithms (дата звернення: 01.06.2026).
6. What is a Container? Docker Inc. URL: https://www.docker.com/resources/what-container/ (дата звернення: 01.06.2026).
7. 10 Docker Myths Debunked. Docker Inc. URL: https://www.docker.com/blog/docker-myths-debunked/ (дата звернення: 01.06.2026).
8. Merkel D. Docker: Lightweight Linux Containers for Consistent Development and Deployment. Linux Journal. 2014. № 239. 5 p.
9. FastAPI. Concurrency and async / await. FastAPI Documentation. URL: https://fastapi.tiangolo.com/async/ (дата звернення: 01.06.2026).

10. SQLite. About SQLite. URL: https://www.sqlite.org/about.html (дата звернення: 01.06.2026).

11. React Team. lazy. React Documentation. URL: https://react.dev/reference/react/lazy (дата звернення: 31.05.2026).

12. Hooks API Reference. React Documentation. URL: https://react.dev/reference/react (дата звернення: 01.06.2026).

13. curl. Command line HTTP. Everything curl. URL: https://everything.curl.dev/http/ (дата звернення: 01.06.2026).

14. Docker. The Docker Ecosystem: An Overview of Containerization. DigitalOcean Tutorials. URL: https://www.digitalocean.com/community/tutorials/the-docker-ecosystem-an-overview-of-containerization (дата звернення: 01.06.2026).

15. Docker Compose overview. Docker Documentation. URL: https://docs.docker.com/compose/ (дата звернення: 01.06.2026).

16. NGINX Documentation. Beginner's Guide. URL: https://nginx.org/en/docs/beginners_guide.html (дата звернення: 01.06.2026).

17. Grafana k6 documentation. Load test types. Grafana Labs. URL: https://grafana.com/docs/k6/latest/testing-guides/test-types/ (дата звернення: 31.05.2026).

18. Сегеда І. В., Беспала О. М. Системи баз даних : комп'ютерний практикум. 2-ге вид. Київ : КПІ ім. Ігоря Сікорського, 2023. URL: https://ela.kpi.ua/server/api/core/bitstreams/149909ad-4995-4913-8e96-56b3795bc9fd/content (дата звернення: 17.05.2026).

19. Microsoft. Create foreign key relationships. Microsoft Learn. URL: https://learn.microsoft.com/en-us/sql/relational-databases/tables/create-foreign-key-relationships (дата звернення: 31.05.2026).

20. Cardellini V., Colajanni M., Yu P. S. Dynamic load balancing on web-server systems. IEEE Internet Computing. 1999. Vol. 3, № 3. P. 28–39.

21. Milanović M. API Gateway vs. Load Balancer vs. Reverse Proxy. Medium. URL: https://medium.com/@techworldwithmilan/api-gateway-vs-load-balancer-vs-reverse-proxy-a04071a767d6 (дата звернення: 17.05.2026).

ДОДАТОК А

Реалізація програмної системи балансування навантаження зворотного проксі-сервера

Програмні засоби:

— балансування, сервіси конвертації – Python;

— навантажувальне тестування – k6;

— контейнеризація – Docker;

— оркестрація – Docker Compose.

Планувальник життєвого циклу контейнерів

```
#!/bin/bash
# scriptA.sh - manages container lifecycle

set -euo pipefail

IMAGE_NAME="tr23malyarchuk/pa-tr23malyarchuk:latest"
NETWORK_NAME="lb-net"
MAX_BUSY_COUNT=2     # minutes before scaling
MAX_IDLE_COUNT=6     # minutes of waiting before stop

# Container metadata
declare -A PORT=( [srv1]=8081 [srv2]=8082 [srv3]=8083 [srv4]=8084 )

container_running() {
    docker ps --format '{{.Names}}' | grep -q "^$1\$"
}

container_exists() {
    docker ps -a --format '{{.Names}}' | grep -q "^$1\$"
}

remove_container() {
    local name=$1
    if container_exists "$name"; then
        echo "[scriptA] Removing container $name..."
        docker rm -f "$name" > /dev/null 2>&1
```

```
    fi
}

launch_container() {
    local name=$1
    local port=${PORT[$name]}
    echo "[scriptA] Starting $name (host port $port)..."

    docker network create "$NETWORK_NAME" 2>/dev/null || true

    docker run -d \
        --name "$name" \
        --network "$NETWORK_NAME" \
        --network-alias "$name" \
        -p "$port:8000" \
        "$IMAGE_NAME" > /dev/null

    echo "[scriptA] $name is up at http://$name:8000 (host port $port)"
}

get_cpu_usage() {
    local name=$1
    docker stats --no-stream --format "{{.CPUPerc}}" "$name" 2>/dev/null \
        | sed 's/%//' | tr -d ' '
}

monitor_until_scaleout_or_idle() {
    local container=$1
    local next=$2
    local busy_count=0
    local idle_count=0

    echo "[scriptA] Monitoring $container..."

    while true; do
        if ! container_running "$container"; then
            echo "[scriptA] $container disappeared. Exiting."
            exit 1
        fi

        local cpu
        cpu=$(get_cpu_usage "$container")

        if [[ -z "$cpu" ]]; then
            echo "[scriptA] Could not read CPU for $container – skipping."
```

```
                sleep 60
                continue
            fi

            echo "[scriptA] $container CPU=${cpu}% (busy=${busy_count} idle=${idle_count})"

            if (( $(echo "$cpu > 30.0" | bc -l) )); then
                busy_count=$(( busy_count + 1 ))
                idle_count=0
            else
                idle_count=$(( idle_count + 1 ))
                busy_count=0
            fi

            if (( idle_count >= MAX_IDLE_COUNT )); then
                echo "[scriptA] System idle. Shutting down."
                shutdown_all
                exit 0
            fi

            if (( busy_count >= MAX_BUSY_COUNT )); then
                echo "[scriptA] $container busy for $MAX_BUSY_COUNT min."
                if [[ -n "$next" ]]; then
                    remove_container "$next"
                    launch_container "$next"
                    echo "[scriptA] Scaled out to $next."
                else
                    echo "[scriptA] At max capacity (srv4)."
                fi
                return
            fi

            sleep 60
        done
    }

    shutdown_all() {
        echo "[scriptA] Stopping all managed containers..."
        for name in "${!PORT[@]}"; do
            if container_running "$name"; then
                docker stop "$name" > /dev/null 2>&1 || true
                docker rm -f "$name" > /dev/null 2>&1 || true
            fi
        done
```

```
    echo "[scriptA] All containers stopped."
}

cleanup_on_interrupt() {
    echo ""
    echo "[scriptA] Caught SIGINT – cleaning up."
    shutdown_all
    exit 0
}

trap cleanup_on_interrupt SIGINT SIGTERM

# Main
echo "[scriptA] Starting up."
remove_container "srv1"
launch_container "srv1"
echo "[scriptA] srv1 running. Press Ctrl+C to stop."

while true; do
    monitor_until_scaleout_or_idle "srv1" "srv2"
    if container_running "srv2"; then
        monitor_until_scaleout_or_idle "srv2" "srv3"
    fi
    if container_running "srv3"; then
        monitor_until_scaleout_or_idle "srv3" "srv4"
    fi
    if container_running "srv4"; then
        monitor_until_scaleout_or_idle "srv4" ""
    fi
    sleep 10
done
```

Сервіси конвертації

```
from fastapi import FastAPI, UploadFile, File, HTTPException
from fastapi.responses import StreamingResponse
from io import BytesIO
import subprocess
from pdf2image import convert_from_bytes
import zipfile
import tempfile
from pathlib import Path
from PIL import Image

app = FastAPI(title="File Converter Service")
```

```
@app.get("/health")
async def health():
    return {"status": "ok"}

@app.post("/convert/wav-to-mp3")
async def wav_to_mp3(file: UploadFile = File(...)):
    data = await file.read()
    with tempfile.NamedTemporaryFile(suffix=".wav", delete=False) as in_f:
        in_f.write(data)
        in_path = in_f.name
    out_path = in_path.replace(".wav", ".mp3")

    try:
        subprocess.check_call(["ffmpeg", "-y", "-i", in_path, out_path])
        with open(out_path, "rb") as f:
            out_bytes = f.read()
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"ffmpeg error: {e}")
    finally:
        for p in [in_path, out_path]:
            try:
                Path(p).unlink(missing_ok=True)
            except Exception:
                pass

    return StreamingResponse(
        BytesIO(out_bytes),
        media_type="audio/mpeg",
        headers={"Content-Disposition": 'attachment; filename="output.mp3"'},
    )


@app.post("/convert/pdf-to-png")
async def pdf_to_png(file: UploadFile = File(...)):
    data = await file.read()
    images = convert_from_bytes(data, dpi=150)
    mem = BytesIO()
    with zipfile.ZipFile(mem, mode="w", compression=zipfile.ZIP_DEFLATED) as zf:
        for idx, img in enumerate(images, start=1):
            buf = BytesIO()
            img.save(buf, format="PNG")
            zf.writestr(f"page_{idx}.png", buf.getvalue())
    mem.seek(0)
    return StreamingResponse(
```

```
            mem,
            media_type="application/zip",
            headers={"Content-Disposition": 'attachment; filename="pages.zip"'},
        )


@app.post("/convert/webp-to-png")
async def webp_to_png(file: UploadFile = File(...)):
    data = await file.read()
    try:
        with Image.open(BytesIO(data)) as im:
            if im.mode not in ("RGB", "RGBA"):
                im = im.convert("RGBA")
            buf = BytesIO()
            im.save(buf, format="PNG")
            buf.seek(0)
            return StreamingResponse(
                buf,
                media_type="image/png",
                headers={"Content-Disposition": 'attachment; filename="output.png"'},
            )
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Conversion error: {e}")


@app.post("/convert/rar-to-zip")
async def rar_to_zip(file: UploadFile = File(...)):
    data = await file.read()
    import rarfile, os, shutil, uuid
    tmp_id = uuid.uuid4().hex
    tmp_dir = Path(f"/tmp/rar_{tmp_id}")
    tmp_dir.mkdir()
    rar_path = tmp_dir / "input.rar"
    try:
        rar_path.write_bytes(data)
        with rarfile.RarFile(rar_path) as rf:
            rf.extractall(tmp_dir / "out")
        mem = BytesIO()
        with zipfile.ZipFile(mem, "w", zipfile.ZIP_DEFLATED) as zf:
            for f in (tmp_dir / "out").rglob("*"):
                if f.is_file():
                    zf.write(f, f.relative_to(tmp_dir / "out"))
        mem.seek(0)
        return StreamingResponse(
            mem,
```

```
            media_type="application/zip",
            headers={"Content-Disposition": 'attachment; filename="output.zip"'},
        )
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"RAR error: {e}")
    finally:
        shutil.rmtree(tmp_dir, ignore_errors=True
```

k6-скрипт

```
import http from 'k6/http';
import { check, sleep } from 'k6';

const wavFile  = open('../data/sample.wav',  'b');
const pdfFile  = open('../data/sample.pdf',  'b');
const webpFile = open('../data/sample.webp', 'b');
const rarFile  = open('../data/sample.rar',  'b');

const BASE_URL = 'http://localhost:8000';

const algorithms = [
  'round_robin',
  'random',
  'least_connections',
  'ip_hash',
  'power_of_two',
];

const ALG_FROM_ENV = __ENV.LB_ALG; // LB_ALG=least_connections k6 run load_test.js

function pickAlgorithm() {
  if (ALG_FROM_ENV && algorithms.includes(ALG_FROM_ENV)) {
    return ALG_FROM_ENV;
  }
  return algorithms[Math.floor(Math.random() * algorithms.length)];
}

export const options = {
  scenarios: {
    wav2mp3_requests: {
      executor: 'per-vu-iterations',
      vus: 20,
      iterations: 50,
      maxDuration: '5m',
```

```
      exec: 'wav2mp3Scenario',
    },
    pdf2png_requests: {
      executor: 'per-vu-iterations',
      vus: 20,
      iterations: 50,
      startTime: '5s',
      maxDuration: '5m',
      exec: 'pdf2pngScenario',
    },
    webp2png_requests: {
      executor: 'per-vu-iterations',
      vus: 20,
      iterations: 50,
      startTime: '10s',
      maxDuration: '5m',
      exec: 'webp2pngScenario',
    },
    rar2zip_requests: {
      executor: 'per-vu-iterations',
      vus: 20,
      iterations: 50,
      startTime: '15s',
      maxDuration: '5m',
      exec: 'rar2zipScenario',
    },
  },
};

// WAV -> MP3
export function wav2mp3Scenario() {
  const algo = pickAlgorithm();
  const res = http.post(`${BASE_URL}/wav2mp3`, {
    file: http.file(wavFile, 'sample.wav', 'audio/wav'),
    algorithm: algo,
  });
  check(res, { 'wav2mp3 status is 200': (r) => r.status === 200 });
  sleep(0.1);
}

// PDF -> PNG
export function pdf2pngScenario() {
  const algo = pickAlgorithm();
  const res = http.post(`${BASE_URL}/pdf2png`, {
    file: http.file(pdfFile, 'sample.pdf', 'application/pdf'),
```

```
    algorithm: algo,
  });
  check(res, { 'pdf2png status is 200': (r) => r.status === 200 });
  sleep(0.1);
}

// WEBP -> PNG
export function webp2pngScenario() {
  const algo = pickAlgorithm();
  const res = http.post(`${BASE_URL}/webp2png`, {
    file: http.file(webpFile, 'sample.webp', 'image/webp'),
    algorithm: algo,
  });
  check(res, { 'webp2png status is 200': (r) => r.status === 200 });
  sleep(0.1);
}

// RAR -> ZIP
export function rar2zipScenario() {
  const algo = pickAlgorithm();
  const res = http.post(`${BASE_URL}/ziprar`, {
    file: http.file(rarFile, 'sample.rar', 'application/vnd.rar'),
    algorithm: algo,
  });
  check(res, { 'rar2zip status is 200': (r) => r.status === 200 });
  sleep(0.1);
}
```

Docker Compose

```
version: '3.8'

networks:
  lb-net:
    driver: bridge
    name: lb-net # docker network

services:
  # Load balancer
  backend:
    build: ./backend
    container_name: lb
    ports:
      - "8000:8000"
```

```
  volumes:
    - /var/run/docker.sock:/var/run/docker.sock
    - ./data:/app/data
  environment:
    - DOCKER_HOST=unix:///var/run/docker.sock
  networks:
    - lb-net
  extra_hosts:
    - "host.docker.internal:host-gateway"
  restart: unless-stopped

#React + NGINX
frontend:
  build: ./frontend
  container_name: frontend
  ports:
    - "80:80"
  networks:
    - lb-net
  depends_on:
    - backend
  restart: unless-stopped
```

ДОДАТОК Б

Апробація результатів роботи

УДК 620.9(062)+621.311(062)
С91

Сучасні проблеми наукового забезпечення енергетики: безпека, стійкість, ІТ та екологічний моніторинг : XXIII Міжнар. наук.-практ. конф. мол. вчених і студ. ; до 40-річчя Чорнобильської катастрофи, м. Київ, 17–22 квіт. 2026 р. – Електрон. текст. дані (1 файл). – Київ : КПІ ім. Ігоря Сікорського, Вид-во «Політехніка», 2026. – 470 с.

ISBN 978-966-990-255-9 (Ел.)

Подано тези доповідей XXIII Міжнародної науково-практичної конференції молодих вчених та студентів «Сучасні проблеми наукового забезпечення енергетики: безпека, стійкість, ІТ та екологічний моніторинг» (до 40-річчя Чорнобильської катастрофи) за такими напрямами: атомна енергетика, ядерна захищеність і нерозповсюдження, теплогідравлічні процеси в тепло- і парогенеруючих установках, сучасні технології в тепловій та альтернативній енергетиці, проблеми теоретичної і промислової теплотехніки, енергетичний менеджмент та інжиніринг, автоматизація енергетичних процесів, стійка енергетика, аспекти розвитку інженерії програмного забезпечення в енергетиці, інформаційні технології та комп'ютерне моделювання.

Подається в авторській редакції за рішенням
Вченої ради Навчально-наукового інституту атомної та теплової енергетики
Національного технічного університету України
«Київський політехнічний інститут імені Ігоря Сікорського»
(протокол № 2 від 30 березня 2026 р.)

ISBN 978-966-990-255-9 (Ел.)

UDC 004.77

[1] BS's programme 4th year Malyarchuk B. V.
[1] Assoc.prof., cand.eng.sc. Demchyshyn A. A.
https://scholar.google.com.ua/citations?user=uHZreb4AAAAJ&hl=uk

[1] Igor Sikorsky Kyiv Polytechnic Institute

REVERSE PROXY LOAD-BALANCING SYSTEM

Statement of the problem and its relevance. In the modern world, users increasingly rely on online services for casual operations, such as image or audio format changes. To provide these services efficiently, backend systems must handle multiple requests concurrently while ensuring fast response times.

Building a system that integrates multiple conversion services under a single access point requires careful consideration of the balancing strategy. Each service has different computational requirements – for example, PDF conversion service is more CPU-intensive than audio format conversion—leading to heterogeneous workloads. An optimal load balancing strategy ensures that all service instances handle requests with minimal response times. In addition, monitoring and administrative interfaces are essential for analyzing resource utilization, and mitigating traffic surges.

Analysis of the latest research. Reverse proxy load balancing has been widely studied and implemented to optimize web service performance and reliability. Prominent software-based solutions include **Nginx**, **HAProxy**, and **Traefik** (a dynamic reverse proxy designed for modern containerized environments with native integration for Docker and Kubernetes). Cloud-oriented offerings, such as **AWS Elastic Load Balancer** and **Azure Application Gateway**, extend reverse proxy load balancing with automated scaling, SSL termination, and health checks.

Containerization technologies such as Docker opened the way for application portability and deployment consistency across diverse computing environments. Research efforts [1, 2] highlight the benefits of containers when compared to traditional virtual machines.

A specific challenge in distributed systems arise when the operated services differ in CPU, memory, and I/O usage characteristics, creating non-uniform workloads. Authors of [3] examine how different request distribution policies influence latency and throughput in multi-tier web services, demonstrating that static round-robin policies may be suboptimal when service execution times vary widely.

Evaluating balancing strategies under controlled conditions has also been a focus of the literature. Traffic generation tools such as Locust, Apache JMeter, and more recently k6 have been used to create synthetic workloads that mimic real-world user behavior, enabling precise measurement of latency, error rates, and resource usage across different configuration regimes. In containerized environments, studies often employ orchestration layers (e.g., Kubernetes, Docker Swarm) and overlay networks to assess how service discovery and internal routing affect end-to-end performance. Metrics collection via Prometheus, Grafana, or custom dashboards is commonly incorporated to provide real-time visualization of system state.

Goal formulation. The problem addressed in this paper is the design and implementation of a reverse-proxy-based load-balancing system for a multi-service web platform that performs file format conversions for multiple simultaneous users. The objective is to evaluate and compare different load balancing strategies at the reverse proxy level and determine which method provides the most efficient request distribution and optimal system performance while maintaining high availability.

Main part. The developed system is a reverse-proxy-based load-balancing system that routes client requests to multiple distributed backend service replicas using NGINX reverse proxy with routing behavior governed by system-controlled parameters. NGINX provides services under a single public IP address and performs TLS termination

The system consists of several coordinated components: configuration server with web-based GUI, a set of docker images hosted at hub.docker.com repository, services front page, k6 load and performance testing scenarios.

Configuration Server with web-based GUI allows users to register services and associate them with specific Docker images, register worker machines, and configure pools of machines available for each service. Metadata is stored in a persistent database. The web interface also provides monitoring and visualization of real-time statistics: machine load, health of running containers, and service latency.

"Services & Images" menu item brings panel where admin registers a new service and defines a respective Docker image name, access port and specifies the period of automatic container update or None for no automatic updates (fig. 1).

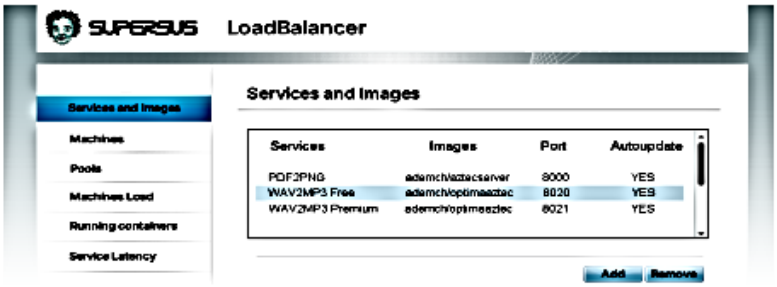


Figure 1 – "Services and Images" panel

"Machines" menu item brings panel for managing (adding and removing) IPs of available computers to run the services on (fig. 2).

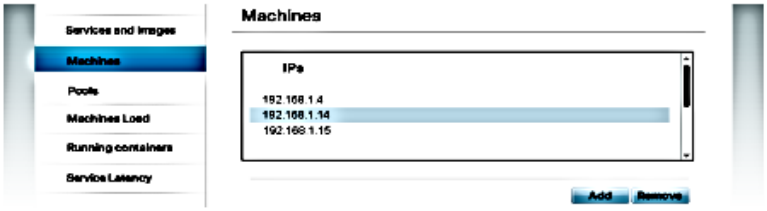


Figure 2 – "Machines" panel

"Pools" menu provides panel for specifying the contents of a pool for each service (fig. 3). It is possible to assign specific machines to the pool or select "Use All Machines" to make the service runnable on all registered computers. The panel contains conditions for containers horizontal growth, whether to start a new container when the number of existing connections is exceeded or the high load continues for more than specified number of minutes.

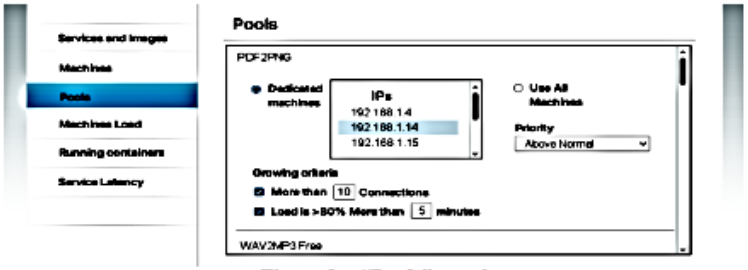


Figure 3 – "Pools" panel

"Machines Load" panel (fig. 4) provides the data as per individual machine helpful for managing existing computational resources.

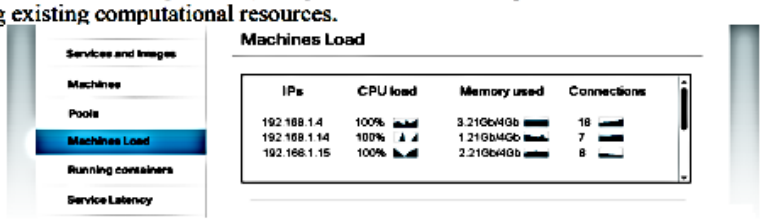


Figure 4 – "Machines Load" panel

"Running containers" panel (fig. 5) visualizes info on working containers and their versions.

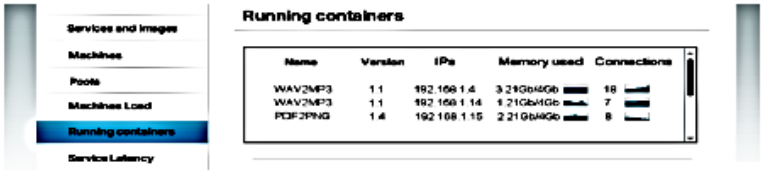


Figure 5 – "Running containers" panel

"Service Latency" panel (fig.6) monitors average completion time that is valuable during estimation of user satisfaction.

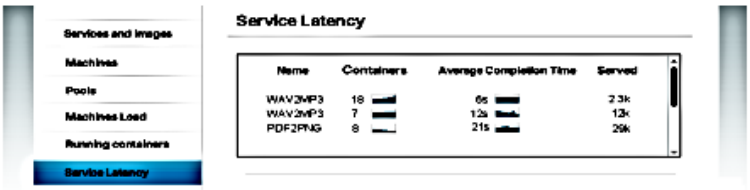


Figure 6 – "Service Latency" panel

User front page. At the front web page, the user can choose the service, upload a document for conversion and download processed result. *A set of docker images.* The backend consists of several containerized conversion services (PDF2PNG, WAV2MP3, and WEBP2PNG), each implemented as an independent Node.js application running in its own Docker container. This architecture enables independent scaling of services, isolation of failures, and supports controlled experimentation with different load balancing strategies. Each service exposes a REST API for file upload and download, as well as endpoints for control and monitoring, including the number of processed requests and the average request processing time.

Docker commands on remote machines are issued over SSH by generating an SSH key with ssh-keygen, copying the public key to the server using ssh-copy-id, and then running commands like ssh user@host docker ps.

Conclusions. Using a least-connections load balancer enhanced with weights and average service latency characteristic resulted in accurate traffic distribution by favoring both higher-capacity nodes and faster-responding instances, resulting in improved overall performance.

References:

1. Bernstein D. Containers and Cloud: From LXC to Docker to Kubernetes. IEEE Cloud Computing. 2014. 1(3). P. 81–84. DOI: 10.1109/MCC.2014.51.

2. Merkel D. Docker: Lightweight Linux Containers for Consistent Development and Deployment. Linux Journal. 2014. Article 2, 239. 5 p.

3. Cardellini V., Colajanni M., Yu P. S. Dynamic load balancing on web-server systems. IEEE Internet Computing. 1999. 3(3), P. 28–39.