Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)
Кафедра комп'ютерних наук
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

бакалавр
(назва освітнього ступеня)

на тему: Розробка серверної системи централізованого збору журналів подій

Виконав: студент IV курсу, групи СНс-41
спеціальності 122 Комп’ютерні науки
(шифр і назва спеціальності)

_____ Дуванов І.П.
(підпис) (прізвище та ініціали)

Керівник _____ Гром'як Р.С.
(підпис) (прізвище та ініціали)

Нормоконтроль _____ Шимчук Г.В.
(підпис) (прізвище та ініціали)

Завідувач кафедри _____ Боднарчук І.О.
(підпис) (прізвище та ініціали)

Рецензент _____ _____
(підпис) (прізвище та ініціали)

Тернопіль
2026

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)

Кафедра комп'ютерних наук
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Боднарчук І.О.
(підпис) (прізвище та ініціали)
«___» _____ 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня Бакалавр
(назва освітнього ступеня)

за спеціальністю 122 Комп'ютерні науки
(шифр і назва спеціальності)

Студенту Дуванову Ігорю Павловичу
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка серверної системи централізованого збору журналів подій

Керівник роботи Гром'як Роман Сильвестрович, к.ф-м.н., доцент кафедри КН
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від «*14*» *травня* *2026* року № *4/9-237*

2. Термін подання студентом завершеної роботи 20 червня 2026р.

3. Вихідні дані до роботи Літературні та інтернет-джерела про журнали подій, централізований збір логів і моніторинг серверних систем. Документація з FastAPI, Redis, REST API, Pydantic, JSON та JSON Lines.

4. Зміст роботи (перелік питань, які потрібно розробити)
Вступ
1) Аналіз завдання та огляд предметної області
2) Реалізація серверної системи централізованого збору журналів подій
3) Тестування роботи серверної системи централізованого збору журналів подій
4) Безпека життєдіяльності, основи охорони праці
Висновки

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)
1. Титульна сторінка. 2. Актуальність дослідження. 3. Мета, об'єкт та предмет дослідження. 4. Завдання дослідження. 5. Журнали подій та їх роль у серверних системах. 6. Загальна архітектура серверної системи. 7. REST API для роботи з журналами подій. 8. Структура журналу події. 9. Зберігання та індексація журналів у Redis. 10. Файловий архів JSON Lines. 11. Підготовка та запуск тестового середовища. 12. Перевірка приймання та зберігання журналів подій. 13. Тестування перегляду, фільтрації та пошуку журналів. 14. Перевірка статистики та останніх подій. 12. Висновки. 13. Завершальний.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Безпека життєдіяльності, основи охорони праці	Мариненко Сергій Юрійович, к.т.н. доцент кафедри МТ		

7. Дата видачі завдання 29 січня 2026 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	Ознайомлення з завданням до кваліфікаційної роботи	30.01.2026	*Виконано*
2.	Підбір джерел по темі кваліфікаційної роботи	31.01.2026-03.02.2026	*Виконано*
3.	Опрацювання джерел по темі кваліфікаційної роботи	04.02.2026-06.02.2026	*Виконано*
4.	Проєктування, реалізація та тестування серверної	07.02.2026-11.02.2026	*Виконано*
	системи централізованого збору журналів подій на		
	основі Redis		
5.	Оформлення розділу «Аналіз завдання та огляд	17.06.2026-23.05.2026	*Виконано*
	предметної області»		
6.	Оформлення розділу « Реалізація серверної системи	24.05.2026-28.05.2026	*Виконано*
	централізованого збору журналів подій»		
6.	Оформлення розділу «Тестування роботи серверної	29.05.2026-04.06.2026	*Виконано*
	системи централізованого збору журналів подій»		
7.	Виконання завдання до розділу «Безпека життєдіяльності»	05.06.2026-06.06.2026	*Виконано*
8.	Виконання завдання до підрозділу «Основи охорони	07.06.2026-08.06.2026	*Виконано*
	праці»		
9.	Оформлення кваліфікаційної роботи	09.06.2026-10.06.2026	*Виконано*
10.	Нормоконтроль	11.06.2026-13.06.2026	*Виконано*
11.	Перевірка на плагіат	14.06.2026	*Виконано*
12.	Попередній захист кваліфікаційної роботи	15.06.2026	*Виконано*
13.	Захист кваліфікаційної роботи	26.06.2026	

Студент _____ Дуванов І.П.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Гром'як Р.С.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Розробка серверної системи централізованого збору журналів подій // Кваліфікаційна робота освітнього рівня «Бакалавр» // Дуванов Ігор Павлович // Тернопільський національний технічний університет імені Івана Пулюя, факультет комп'ютерно-інформаційних систем і програмної інженерії, кафедра комп'ютерних наук, група СНс-41 // Тернопіль, 2026 // С.83, рис. – 51, табл. – 1, кресл. – 16, додат. – 0, бібліогр. – 35.

Ключові слова: Redis, FastAPI, REST API, JSON, JSON Lines, filtering, statistics, file archive.

У кваліфікаційній роботі бакалавра розглянуто питання розроблення серверної системи централізованого збору журналів подій. Метою дослідження було проєктування, реалізація та тестування програмного рішення для приймання, зберігання, перегляду, фільтрації та статистичної обробки логів від різних сервісів. У роботі проаналізовано роль журналів подій у роботі серверних систем, підходи до централізованого збору та зберігання логів, а також існуючі програмні рішення для моніторингу й аналізу журналів. Практична частина передбачала створення серверної системи на основі FastAPI з використанням Redis як основного сховища даних. Було реалізовано REST API для приймання одного журналу або пакета журналів, перегляду подій, пошуку за рівнем критичності, назвою сервісу та часовим проміжком. Також у системі реалізовано отримання останніх подій, формування статистики та додаткове збереження журналів у файловому архіві формату JSON Lines. Результатом роботи стало функціональне програмне рішення, яке дозволяє централізовано збирати журнали подій від різних сервісів, швидко зберігати їх у Redis, виконувати пошук і фільтрацію, аналізувати загальну кількість подій та контролювати роботу серверної системи. Проведене тестування підтвердило коректність роботи основних функцій системи та доцільність її використання для моніторингу вебсервісів, мікросервісів і серверних застосунків.

ANNOTATION

Development of a Server-Side Centralized Event Log Collection System // Qualification work of the educational level "Bachelor" // Duvanov Igor // Ternopil Ivan Pulyu National Technical University, Computer and Information Systems and Software Engineering Faculty, Computer Sciences Department, group SNs-41 // Ternopil, 2026 // P. 83, fig. - 51, tabl. - 1, drawings - 16, annexes. – 0, references - 35.

Keywords: Redis, FastAPI, REST API, JSON, JSON Lines, filtering, statistics, file archive.

The bachelor's qualification thesis considers the development of a server system for centralized collection of event logs. The purpose of the research was to design, implement, and test a software solution for receiving, storing, viewing, filtering, and statistically processing logs from different services. The thesis analyzes the role of event logs in server systems, approaches to centralized log collection and storage, as well as existing software solutions for monitoring and log analysis. The practical part involved the creation of a server system based on FastAPI using Redis as the main data storage. A REST API was implemented for receiving a single log event or a batch of log events, viewing events, and searching by severity level, service name, and time period. The system also provides retrieval of the latest events, generation of statistics, and additional storage of logs in a JSON Lines file archive. As a result, a functional software solution was developed that allows centralized collection of event logs from different services, fast storage in Redis, search and filtering, analysis of the total number of events, and monitoring of the server system operation. The conducted testing confirmed the correctness of the main system functions and the feasibility of using it for monitoring web services, microservices, and server applications.

ПЕРЕЛІК СКОРОЧЕНЬ

API (англ. Application Programming Interface) - програмний інтерфейс застосунку.

REST (англ. Representational State Transfer) - архітектурний стиль взаємодії між клієнтом і сервером.

JSON (англ. JavaScript Object Notation) - текстовий формат обміну структурованими даними.

Redis (англ. Remote Dictionary Server) - сховище даних типу «ключ-значення», що працює в оперативній пам’яті.

DB (англ. Database) - база даних.

CLI (англ. Command Line Interface) - інтерфейс командного рядка.

ILM (англ. Index Lifecycle Management) - керування життєвим циклом індексів.

ЗМІСТ

ВСТУП

Актуальність теми. В умовах активного розвитку вебсервісів, мікросервісної архітектури, хмарних платформ та розподілених інформаційних систем особливої актуальності набуває питання централізованого збору, зберігання та аналізу журналів подій. Сучасні серверні системи складаються з великої кількості окремих сервісів, кожен із яких формує власні повідомлення про помилки, попередження, інформаційні події, дії користувачів та внутрішні процеси. Якщо такі журнали зберігаються окремо в різних частинах системи, їх складно аналізувати, порівнювати між собою та використовувати для швидкого виявлення проблем.

Журнали подій є важливим джерелом інформації про роботу програмної системи, оскільки вони дозволяють відстежувати помилки, аналізувати поведінку сервісів, контролювати критичні події, виконувати аудит і швидше реагувати на інциденти. Особливо важливим є не лише збереження журналів, а й можливість їх фільтрації, пошуку за часовим проміжком, рівнем критичності, назвою сервісу та іншими параметрами. У зв’язку з цим актуальним є розроблення серверної системи централізованого збору журналів подій, яка забезпечує приймання логів від різних сервісів, їх обробку, швидке зберігання в Redis, формування статистики та додаткове архівування у файловому сховищі.

Мета і задачі дослідження. Метою даної роботи є проєктування, реалізація та тестування серверної системи централізованого збору журналів подій на основі Redis з додатковим файловим архівом, яка забезпечує приймання, зберігання, перегляд, фільтрацію, пошук і статистичну обробку логів від різних сервісів.

Для досягнення поставленої мети було сформульовано такі основні задачі:

– дослідити призначення та роль журналів подій у серверних системах;

– дослідити підходи до централізованого збору, зберігання та обробки журналів;

– спроєктувати архітектуру серверної системи централізованого збору журналів подій;

– реалізувати REST API для приймання, перегляду, фільтрації та пошуку журналів;

– реалізувати зберігання журналів у Redis та додатковому файловому сховищі;

– забезпечити отримання статистики, останніх подій і перевірити роботу системи під час тестування.

Об’єкт дослідження. Об’єктом дослідження є процеси централізованого збору, зберігання та обробки журналів подій у сучасних серверних інформаційних системах.

Предмет дослідження. Предметом дослідження є методи, засоби та програмні механізми приймання, збереження, індексації, фільтрації, пошуку, архівування та статистичної обробки журналів подій із використанням FastAPI, Redis і файлового сховища.

Практичне значення одержаних результатів. Практичне значення роботи полягає у розробці серверної системи централізованого збору журналів подій, яку можна використовувати для моніторингу вебсервісів, мікросервісів та інших серверних застосунків. Реалізоване рішення дозволяє приймати журнали через REST API, зберігати їх у Redis, фільтрувати за рівнем критичності, сервісом і часом, отримувати останні події, формувати статистику та виконувати резервне копіювання записів у файлове сховище.

Запропонований підхід спрощує пошук помилок, аналіз роботи системи та контроль критичних подій. Отримані результати можуть бути використані для подальшого розвитку систем моніторингу, інтеграції зі сповіщеннями та візуалізацією логів, а також у навчальних цілях.

РОЗДІЛ 1. АНАЛІЗ ЗАВДАННЯ ТА ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Поняття журналів подій та їх роль у роботі серверних систем

У сучасних серверних інформаційних системах журнал подій є не просто допоміжним технічним файлом, а одним з базових носіїв операційної пам’яті інфраструктури. У фундаментальному визначенні журнал розглядають як запис подій, що відбуваються в системах і мережах організації, причому кожен запис пов’язаний з конкретною подією, яка вже сталася. У міру розвитку мережевих серверних середовищ роль журналів істотно розширилася. Вони використовуються не лише для локального усунення несправностей, а і для оптимізації продуктивності, фіксації дій користувачів, виявлення злочинної активності, аудиту, цифрової криміналістики та підтримки реагування на інциденти. Саме тому керування журналами в науковій і нормативній літературі розглядається як окремий процес, що охоплює генерування, передавання, зберігання, аналіз і контрольоване видалення журналів [1,2].

Для серверних систем значення журналів подій особливо велике, оскільки саме сервери забезпечують безперервне функціонування прикладних сервісів, обслуговують мережеву взаємодію та накопичують відомості про адміністративні й користувацькі дії [3]. Коли користувач автентифікується, прикладна служба виконує запит, база даних повертає помилку, міжмережевий екран відхиляє пакет або система оркестрації переносить контейнер на інший вузол, саме журнали стають головним джерелом реконструкції послідовності подій. NIST прямо вказує, що до найбільш важливих категорій належать журнали засобів безпеки, журнали операційних систем і журнали застосунків, а також наголошує, що в реальній інфраструктурі один і той самий інцидент часто відображається одразу в кількох різних джерелах з різною повнотою та деталізацією [1]. Звідси випливає, що для серверної системи журналювання

ключовою є не сама наявність записів, а можливість зіставити їх між собою в єдиному часовому та семантичному контексті.

Класифікація журналів подій у предметній області має практичний характер і зазвичай будується навколо джерела, змісту та режиму використання. За джерелом доцільно виділяти журнали операційної системи, прикладні журнали, журнали мережевих пристроїв, журнали засобів захисту, журнали баз даних, вебсерверів, контейнерних платформ і хмарних сервісів. За змістом це можуть бути журнали автентифікації та авторизації, аудиту дій, системних помилок, транзакцій, доступу, продуктивності та безпекових спрацювань. За рівнем структурованості журнали подій поділяють на неструктуровані текстові, напівструктуровані та повністю структуровані. Історично важливу роль в уніфікації журналювання відіграв стандарт syslog, який визначив узагальнений формат повідомлень журналу та механізми їх передавання. У сучасних реалізаціях syslog передбачає відокремлення змісту повідомлення від транспортного рівня, а також підтримує розширення повідомлень за допомогою структурованих даних [4]. У практиці централізованого збору це означає, що придатність журналу до подальшого аналізу визначається не лише змістом повідомлення, а й ступенем формалізації його полів, наявністю часової мітки, ідентифікатора джерела, рівня серйозності, контексту процесу й технічних атрибутів, придатних до машинної обробки [4].

Транспортний аспект журналювання є не менш важливим, ніж змістовий. Для syslog документ RFC 5426 описує передавання повідомлень журналу через UDP і водночас прямо вказує на суттєві проблеми надійності та безпеки такого способу передавання. До них належать відсутність механізмів підтвердження й повторної доставки, можливість втрати датаграм, відсутність конфіденційності під час передавання, ризик підроблення та повторного відтворення повідомлень, а також неможливість надійно встановити послідовність подій лише за порядком їх надходження [5]. У тому самому документі наголошується, що загальне застосування має орієнтуватися на TLS транспорт, а не на UDP в незахищених мережах [5]. RFC 5425, своєю чергою, визначає

застосування TLS для syslog і підкреслює, що цей механізм призначений для створення захищеного з’єднання та протидії типовим загрозам безпеці syslog [6]. Для серверної системи централізованого збору з цього випливає принципове проєктне правило, згідно з яким вибір транспорту не може розглядатися як суто мережеве налаштування. Це рішення безпосередньо впливає на достовірність журналів, доказову цінність записів і придатність даних для подальшого аудиту та реагування на інциденти [6,7].

У розподілених серверних середовищах додатковою проблемою є різнорідність форматів журналів і відсутність спільної моделі подій. Саме на подолання цієї проблеми орієнтовано сучасні специфікації OpenTelemetry. У моделі журналів OpenTelemetry визначено узгоджене розуміння запису журналу, а також даних, які мають бути зафіксовані, передані, збережені та інтерпретовані системою журналювання. Специфікація передбачає можливість однозначного відображення наявних форматів журналів до спільної моделі зі збереженням семантики їхніх основних елементів. У межах цієї моделі виділяють такі суттєві поля, як час події, час спостереження, рівень серйозності, тіло повідомлення, ресурс, атрибути події, а також поля трасувального контексту TraceId і SpanId, що дає змогу пов’язувати журнали з трасами розподілених запитів [8]. Водночас протокол OTLP описує кодування, транспорт і механізми доставки телеметричних даних між джерелами, проміжними вузлами та бекендами, а OpenTelemetry Collector забезпечує незалежний від конкретного постачальника механізм приймання, обробки та експорту телеметрії, який може розгортатися як агент або як шлюз [9]. Для предметної області це означає перехід від ізольованого збирання текстових записів до побудови універсального каналу телеметрії, у межах якого журнали, метрики та траси дедалі частіше розглядаються як взаємопов’язані сигнали [10].

Саме тому роль журналів у роботі серверних систем сьогодні доцільно тлумачити у трьох взаємопов’язаних площинах. У першій площині журнали є операційним інструментом спостережуваності, який дозволяє встановити, що саме сталося, де, коли і за яких умов. У другій площині вони виконують

безпекову та контрольну функцію, оскільки фіксують спроби доступу, зміни конфігурації, мережеві аномалії та відхилення від політик. У третій площині журнали стають аналітичним джерелом для побудови кореляцій, статистики, сповіщень і трендів. Ця третя площина можлива лише за умови нормалізації, тобто приведення різнорідних записів до спільного набору полів. Цю ідею формалізує Elastic Common Schema, яка визначає спільний набір полів для подій, зокрема журналів і метрик, і прямо вказує, що метою ECS є нормалізація подій заради кращого аналізу, візуалізації та кореляції [11]. Отже, в сучасній серверній архітектурі журнал подій перестає бути другорядним текстом і стає стандартизованим носієм спостережуваних фактів, на основі яких працюють механізми пошуку, моніторингу, безпекового аналізу та оперативного реагування [11].

1.2 Аналіз підходів до централізованого збору, зберігання та обробки журналів подій

Централізований збір журналів виникає як відповідь на фундаментальну проблему фрагментації джерел. NIST підкреслює, що в типовій організації журнали створюються великою кількістю вузлів, а їхня розрізненість, неоднорідність і відмінності у часових мітках ускладнюють аналіз та зіставлення подій [1]. У відповідь на це формується інфраструктурний підхід, у якому перший рівень складається з генераторів журналів, другий рівень виконує приймання, аналіз і зберігання, а третій рівень забезпечує моніторинг, перегляд і звітність. У межах такої архітектури дані можуть передаватися в реальному або майже реальному часі, а також пакетами за розкладом. Важливо, що NIST допускає як модель, де клієнти самі надсилають записи на сервер збору, так і модель, у якій центральний вузол автентифікується на джерелі й отримує копії журналів. Вже на цьому етапі видно базовий компроміс між простотою, оперативністю, керованістю та навантаженням на мережу [1].

Звідси випливають ключові вимоги до серверної системи централізованого журналювання. Вона має бути надійною, тобто не втрачати записи під час пікових навантажень або відмов окремих компонентів. Вона має бути масштабованою, щоб приймати дедалі більший обсяг подій без деградації та затримки. Вона має бути безпечною, тобто захищати журнали в транзиті та у спокої, обмежувати доступ до них і зберігати цілісність. Вона має підтримувати різні формати джерел і способи доставки, бо в реальному середовищі одночасно співіснують syslog, локальні файлові журнали, події застосунків, JSON повідомлення, хмарні журнали та телеметрія від агентів. Нарешті, вона повинна бути придатною до нормалізації, оскільки без спільних полів централізація перетворюється на просте накопичення різнорідного тексту без аналітичної цінності [11].

На концептуальному рівні підходи до централізованого збору можна умовно поділити на файл-орієнтовані, потік-орієнтовані та телеметричні [12,13]. У файл-орієнтованій схемі спеціальний процес читає локальні журнальні файли та пересилає їх далі. У потік-орієнтованій схемі джерело одразу надсилає події до колектора, наприклад через syslog або HTTP в потоковому режимі. Телеметричний підхід опирається на структуровану модель сигналів, у якій лог уже на момент генерації описаний через формалізовані поля та може бути переданий універсальним протоколом до проміжного колектора або бекенда [14]. Практична цінність третього підходу полягає у тому, що значна частина робіт з подальшої обробки переноситься ближче до джерела або виконується в колекторі ще до індексації, що спрощує пов’язування записів журналу з ланцюгами виконання розподілених запитів (трасами) і відповідними метриками. Водночас у серверних інфраструктурах найчастіше співіснують усі три підходи, оскільки застарілі мережеві пристрої продовжують надсилати syslog, сучасні застосунки формують JSON або OTel журнали, а системні агенти читають локальні файли.

Наступним критично важливим етапом є зберігання та життєвий цикл даних. Для високонавантажених журналів домінує модель часових потоків, де

нові події постійно дописуються, а старі переходять до дешевшого шару зберігання. В екосистемі Elastic це реалізується через data streams, які розглядаються як абстракція над набором прихованих backing indices для append only часових даних, придатних для журналів, подій і метрик. ILM автоматизує rollover, retention і видалення часових індексів, а політика життєвого циклу може переносити старіші індекси на дешевше обладнання. На рівні фізичної організації Elastic виділяє гарячі, теплі, холодні та frozen шари з різними характеристиками доступу й вартості. Додатково важливо, що сучасний logsdb режим для logs data streams орієнтований на ефективніше зберігання журналів і в бенчмарках показує зменшення просторового сліду, хоча абсолютний ефект залежить від конкретного набору даних [14]. На практиці це означає, що сучасна централізована система має розглядати журнальні дані не як єдиний монолітний файл, що безперервно зростає, а як потік часових сегментів з окремими правилами індексації, пошуку, стискання та видалення.

Інший принцип реалізує Grafana Loki, де вміст журнальних рядків не індексується повністю. Індекс охоплює лише метадані у вигляді міток, які групують записи у журнальні потоки. Самі журнальні рядки стискаються та зберігаються окремими блоками даних, зазвичай в object storage, а індекс вказує, у яких блоках містяться записи з відповідним набором міток. У документації Loki окремо наголошується, що така модель з малим індексом і сильно стиснутими блоками даних спрощує експлуатацію та знижує вартість зберігання. Для керування строками зберігання у Grafana Loki використовується компонент Compactor, який оптимізує структуру індексів та застосовує політики збереження журналів. У сучасних режимах зберігання видалення даних відбувається у два етапи. Спочатку з індексу вилучаються посилання на застарілі блоки журнальних даних, а потім ці блоки асинхронно видаляються зі сховища [15]. Звідси постає важлива відмінність між документно-пошуковими та label oriented архітектурами. Перші краще підходять для довільного повнотекстового пошуку і складної польової

аналітики, другі є дуже ефективними, коли основний сценарій полягає у відборі потоків за стабільними мітками та подальшому аналізі тексту в межах відносно вузької вибірки [16]. Саме тому проєктування схеми міток і недопущення надмірної кількості комбінацій їхніх значень стає для Loki таким самим значущим, як проєктування mappings і шаблонів індексів для систем на базі Elastic чи OpenSearch [17].

Збір і зберігання без обробки не дають повної цінності. Тому друга велика група підходів пов’язана з попереднім опрацюванням, нормалізацією, збагаченням і кореляцією. У Graylog вже на етапі входу логічні inputs приймають повідомлення з різних джерел і протоколів, причому документація розрізняє listener і pull based inputs. Далі Processing Pipelines дозволяють перевіряти умови, трансформувати, збагачувати та маршрутизувати дані ще до збереження або індексації, а event definitions будують над журналами модель подій і сповіщень [18]. У Wazuh аналогічний ланцюг має більш виражену безпекову спрямованість. Система може збирати дані через агенти, agentless monitoring, syslog та API, після чого analysis engine виконує decoding, rule matching і генерує alerts, які потім індексуються та візуалізуються [19]. Сутнісно йдеться про перехід від пасивного накопичення сирих записів до активного перетворення журналу в подію з атрибутами, контекстом, пріоритетом і реакцією. У серверній системі централізованого збору саме цей крок визначає, чи стане сховище журналів аналітичною платформою, чи залишиться просто архівом тексту [19].

Для обробки неструктурованих журналів ключове значення має парсинг, тобто опрацювання журнальних повідомлень і виокремлення з них структурованих полів. У документації Elastic grok processor прямо визначено як механізм витягування структурованих полів із текстового поля документа за шаблонами, а в сумісному з ECS режимі імена полів одразу наближаються до спільної схеми подій [11]. У світі OpenSearch схожу функцію виконує Data Prepper з Grok Processor, який приймає журнали, наприклад від Fluent Bit через HTTP source, і структурує їх до індексації в OpenSearch [20]. У Wazuh ту саму

функцію в безпековому домені виконують decoders, які виділяють значущі поля та готують записи до rule matching [19]. Отже, парсинг слід розглядати не як локальну утиліту для “красивішого” подання повідомлення, а як обов’язкову передумову нормалізації, точного пошуку, кореляції та побудови сигнатурних або аналітичних правил. Чим раніше сирий рядок перетворюється на структуру полів, тим меншим є навантаження на запити, агрегації та механізми оповіщення.

Важливою умовою надійності централізованого збору є буферизація та запобігання втраті даних у разі перевантаження приймача або тимчасової недоступності сховища. Fluent Bit прямо вказує, що пропонує кілька режимів буферизації, серед яких memory only, memory ring buffer і filesystem buffering. Документація наголошує, що memory only режим швидший, але більш схильний до втрати даних, тоді як файлова буферизація менш ефективна, однак менш схильна до втрати. Окремо зазначено, що в умовах backpressure деякі input plugins при досягненні mem_buf_limit можуть втрачати дані, і саме тому файлові буфери є кращими, коли втрата неприпустима [20]. Якщо ж між шаром збору та шаром індексації необхідно створити додатковий бар’єр відмовостійкості, практичним рішенням стає брокер повідомлень на кшталт Apache Kafka. У документації Kafka підкреслюється, що acks=all дає найсильнішу гарантію збереження запису, поки живою залишається хоча б одна in sync replica, а idempotence дозволяє гарантувати запис рівно однієї копії повідомлення в потік [21]. Для серверної системи централізованого журналювання це означає, що архітектура без буферів і без керування перевантаженням є вразливою до піків інтенсивності та відмов сховища, тоді як наявність багаторівневого буферування перетворює конвеєр збору на стійку асинхронну систему [21].

Окремо слід розглянути питання архітектури сховища та індексації. В OpenSearch для роботи з потоковими часовими даними, зокрема журналами, застосовуються data streams і механізми Index State Management, які дають змогу керувати життєвим циклом індексів та переводити їх між етапами hot,

warm і delete. У такій архітектурі Data Prepper може виконувати роль масштабованого проміжного шару для приймання, попередньої обробки та передавання журнальних даних [22]. Splunk реалізує іншу, але концептуально близьку модель керування життєвим циклом індексованих даних. У цій моделі дані організовуються в сегменти, які переходять між станами hot, warm, cold, frozen і thawed. Гарячі сегменти використовуються для активного запису, теплі залишаються доступними для пошуку без подальшого дописування, холодні можуть переноситися на дешевше сховище, а заморожені за замовчуванням видаляються або архівуються з можливістю подальшого відновлення [23]. Якщо порівняти ці архітектури, то можна побачити спільну логіку. Усі зрілі платформи намагаються розвести швидкий прийом свіжих даних, ефективний пошук по відносно нових сегментах, дешеве зберігання старих даних і контрольоване видалення або архівування. Різниця між ними полягає в тому, як саме вони реалізують індексацію, з якою дсикретністю оперують часовими сегментами та які типи запитів оптимізують у першу чергу.

Під час централізованого збору журналів не можна ігнорувати регуляторні та безпекові вимоги. Загальний регламент ЄС про захист даних GDPR закріплює принципи обмеження мети обробки, мінімізації даних, обмеження строку зберігання, цілісності та конфіденційності, а також вимагає враховувати захист даних за задумом і за замовчуванням [24]. Для журналів подій це має безпосереднє значення, оскільки журнальні записи часто містять ідентифікатори користувачів, імена облікових записів, IP адреси, заголовки HTTP, значення параметрів або службові повідомлення, які можуть опосередковано або безпосередньо стосуватися персональних даних. Отже, централізована система має підтримувати керування строками зберігання, маскування чутливих атрибутів, сегментацію доступу, шифрування в транзиті, контроль цілісності та аудит дій над самими журналами. Цей висновок узгоджується і з рекомендаціями NIST щодо обмеження доступу до журналів, перевірки їх цілісності та захисту мережевих комунікацій, пов’язаних з лог даними [1].

Практичне розгортання централізованого збору журналів завжди пов’язане з вибором між агентним і безагентним підходами до передавання даних. Агентний підхід забезпечує кращий контроль локальних журнальних файлів, дає змогу буферизувати записи поруч із джерелом і виконувати попередню нормалізацію ще до мережевого передавання. Безагентний підхід зменшує операційні витрати на підтримку великої кількості клієнтських компонентів, однак більшою мірою залежить від мережевих протоколів, доступності сервісів і обмежень самого джерела даних. У сучасних системах обидва підходи співіснують. OpenTelemetry Collector може працювати як агент або шлюз [9], Graylog підтримує як приймання повідомлень від зовнішніх джерел, так і самостійне отримання даних з визначених ресурсів [18], Wazuh поєднує агентний моніторинг із безагентним контролем систем [19], а NIST на рівні архітектури допускає як мережеву публікацію журналів, так і централізоване отримання копій журнальних файлів [1]. На цьому фоні високодоступне розгортання потребує резервування колекторів, дублювання критичних вузлів індексації, рознесення функціональних ролей на окремі сервери, перевірених процедур архівування та відновлення, а також регулярного контролю коректності ротації журналів. Таку логіку підтримують багатовузлові моделі Graylog і Wazuh, архівні життєві цикли Splunk, а також архітектури Loki та Elastic, орієнтовані на використання об’єктного сховища.

1.3 Огляд існуючих програмних рішень для моніторингу та аналізу логів

Сучасний ринок систем збирання та аналізу журналів складається не з одного універсального класу платформ, а з кількох архітектурних сімейств. Одну групу становлять рішення, орієнтовані на повнотекстовий пошук і аналіз записів подій з урахуванням їхніх структурованих атрибутів. До цієї групи належать Elastic Stack, OpenSearch та Splunk. Інша група рішень спирається на потокову модель, у якій індексується лише обмежений набір метаданих, а

основний обсяг текстових журнальних даних зберігається в об’єктному сховищі з нижчою вартістю зберігання. Такий підхід реалізовано в Loki [15]. Окрему групу становлять системи з вираженою безпековою орієнтацією, які доповнюють збирання журналів механізмами декодування, правилами виявлення, сповіщеннями та формуванням подій SIEM. Це характерно для Wazuh і частково для Graylog. Тому коректне порівняння наявних рішень потребує уточнення цільового сценарію, оскільки одна й та сама система може бути ефективною для аналітики за інцидентами, але менш зручною для довготривалого зберігання журналів з мінімальними витратами, або навпаки [23].

Elastic Stack і пов’язані з ним інструменти моніторингу мають одну з найбільш розвинених концепцій керування життєвим циклом журнальних даних. Сильним боком цієї екосистеми є поєднання кількох рівнів стандартизації та оптимізації. ECS нормалізує структуру подій, grok processor дає змогу перетворювати текстові повідомлення на набір полів, data streams організовують часові дані, що лише дописуються, в єдиний логічний ресурс, ILM автоматизує перехід на нові індекси, керування строками зберігання та видалення, а рівні зберігання даних розмежовують продуктивне зберігання і зберігання з нижчою вартістю. Така сукупність можливостей робить Elastic особливо придатним для сценаріїв, у яких від системи очікують не лише пошуку рядків, а й аналізу за різними полями, агрегування та пов’язування подій за службами, хостами, користувачами чи контекстом виконання запиту.. Водночас така функціональна насиченість підвищує вимоги до проєктування схем полів, шаблонів індексів, політик життєвого циклу та контролю обсягу даних. Іншими словами, Elastic забезпечує високу аналітичну гнучкість у масштабованих середовищах, але потребує складнішої експлуатації, що є типовим компромісом для функціонально насичених пошукових платформ.

Grafana Loki, навпаки, демонструє підхід економної індексації. Система індексує лише мітки, а сам текст журналів стискає в окремі блоки даних і розміщує в об’єктному сховищі. Для середовищ, у яких критично важливим є

масштабоване й відносно недороге довготривале зберігання великих обсягів журналів, така модель часто є привабливою [15]. Особливо природно Loki вбудовується в екосистему хмарних застосунків, де мітками можуть бути namespace, service, pod, cluster або інші стабільні атрибути. Проте саме тут проходить головна межа придатності цього підходу. Оскільки повний текст журналів не є основним простором індексування, якість запитів суттєво залежить від вдалого проєктування міток і винесення атрибутів з великою кількістю унікальних значень до структурованих метаданих. Документація Loki окремо підкреслює, що структуровані метадані корисні саме для таких атрибутів, оскільки їх індексація як міток була б надто витратною [15]. Відповідно, Loki можна вважати сильним рішенням для масштабного операційного журналювання, однак воно потребує високої дисципліни у проєктуванні семантики міток і не завжди є оптимальним там, де переважає довільний повнотекстовий пошук за великою кількістю полів [15].

Graylog займає проміжне місце між класичними системами керування журналами та більш спеціалізованими безпековими платформами. Важливо, що документація Graylog Open прямо описує продукт як безоплатне відкрите рішення для централізованого керування журналами, здатне збирати, та аналізувати дані з різних середовищ [18]. З архітектурного погляду Graylog привабливий тим, що досить прозоро розділяє приймання даних, конвеєри обробки, потоки повідомлень, правила формування подій, сповіщення та варіанти розгортання від базового до високодоступного режиму [18]. Завдяки цьому він добре підходить для організацій, яким потрібна самостійно керована платформа з гнучкою маршрутизацією, правилами обробки та зручним інтерфейсом роботи з журналами, але без обов’язкового переходу до надмірно складного комплексу моніторингу й аналізу телеметрії. Обмеження Graylog полягають не стільки у функціональності, скільки в тому, що зі зростанням обсягу даних організація все одно стикається з потребою проєктувати багатокомпонентне розгортання, окремо враховувати Data Nodes, MongoDB, розподіл ролей і продуктивність пошуку [18]. Тобто Graylog є сильною

системою централізованого журналювання, але, як і всі зрілі платформи, потребує архітектурної дисципліни під час експлуатації [18].

Wazuh вирізняється чіткою безпековою спеціалізацією. В офіційній документації він визначається як безоплатна відкрита безпекова платформа, що поєднує можливості XDR та SIEM і працює з локальними, віртуалізованими, контейнеризованими та хмарними середовищами [19]. На відміну від більш загальних платформ журналювання, Wazuh розглядає журнал передусім як джерело виявлення загроз, а не лише як матеріал для пошуку. Збирання даних через агентні та безагентні механізми, декодери, правила, формування сповіщень, індексація у власному індексаторі та подальша візуалізація на інформаційній панелі утворюють безперервний цикл виявлення й первинного оцінювання інцидентів [19]. Це робить Wazuh привабливим у середовищах, де першочергове значення мають безпековий моніторинг, контроль цілісності, відповідність політикам і правила кореляції, характерні для SIEM. Разом з тим для суто операційного журналювання загального призначення, де потрібні довільні аналітичні сценарії, широке прикладне спостереження та глибока інтеграція з практиками моніторингу й аналізу телеметрії, Wazuh доцільніше розглядати як безпекове ядро, а не як повний замінник усіх типів платформ для роботи з журналами [19].

OpenSearch у поєднанні з Data Prepper репрезентує напрям відкритих масштабованих пошуково-аналітичних платформ, здатних забезпечити повний ланцюг роботи з журналами від приймання даних до керування їхнім життєвим циклом [22]. Його сильна сторона полягає у близькості до підходів сімейства Elasticsearch при збереженні самостійної екосистеми. Data streams, керування станами індексів через ISM, можливість реалізації сценаріїв із переходом між станами hot, warm і delete, а також наявність Data Prepper як масштабованої проміжної ланки приймання та попередньої обробки даних створюють достатньо зрілу базу для побудови централізованого журналювання [22]. Водночас OpenSearch часто вимагає від команди більшої самостійності у виборі компонентів, шаблонів, процесорів обробки вхідних даних та експлуатаційних

практик. Це не є недоліком як таким, а радше відображає інший профіль інженерної відповідальності. Платформа дає багато свободи, але успіх проєкту тут більше залежить від кваліфікації команди, яка проєктує схему даних, життєвий цикл індексів і правила обробки [22].

Splunk Enterprise традиційно розглядають як один з еталонів корпоративного середовища для централізованого збирання, пошуку та аналізу журналів. Офіційні матеріали Splunk описують продукт як платформу, що дає змогу шукати, аналізувати та візуалізувати дані з усього технологічного середовища, а технічна документація детально формалізує життєвий цикл сегментів індексованих даних, механізми архівування та подальшого відновлення [23]. Архітектурна сила Splunk полягає в чітко опрацьованій моделі старіння даних, у якій стани hot, warm, cold і frozen зрозумілі з погляду продуктивності, вартості зберігання й експлуатаційної практики [23]. Для великих організацій привабливою є як зрілість екосистеми, так і глибока опрацьованість сценаріїв експлуатації. Водночас важливо враховувати, що Splunk орієнтований на корпоративну модель використання, тому є доцільним насамперед там, де обсяг даних, організаційні процеси, рівень зрілості команди та вимоги до підтримки виправдовують вибір великої комерційної платформи [23]. У підсумку Splunk варто порівнювати не з окремими утилітами збирання журналів, а з повноцінними платформами централізованої пошуково-аналітичної інфраструктури [23].

Якщо узагальнити порівняння, то Elastic і OpenSearch є сильними там, де потрібні аналіз за різними полями, гнучкі схеми даних і керування життєвим циклом часових даних. Loki має переваги там, де основною проблемою є висока вартість індексації надвеликих потоків журналів і де запити природно описуються через мітки та часові вікна [15]. Graylog забезпечує зручний баланс між централізованим керуванням журналами, конвеєрами обробки та механізмами сповіщення [18]. Wazuh найбільш доречний у сценаріях, орієнтованих насамперед на безпеку [19]. Splunk є сильним представником зрілих корпоративних платформ з відпрацьованою моделлю зберігання та

пошуку [23]. Тому вибір конкретного рішення має ґрунтуватися не на популярності інструмента, а на характері журналів, цільовій моделі запитів, вимогах до безпеки, ресурсній моделі зберігання, потрібній швидкості пов’язування подій і готовності організації підтримувати відповідну складність платформи.

1.4 Висновок до першого розділу

У першому розділі проаналізовано теоретичні засади функціонування журналів подій та визначено їхню роль у роботі сучасних серверних систем. Встановлено, що журнали є важливим джерелом інформації про стан системи, дії користувачів, помилки, мережеві взаємодії, безпекові інциденти та роботу окремих сервісів. Їх використання не обмежується пошуком помилок, оскільки дані з журналів також застосовуються для аудиту, моніторингу, аналізу продуктивності, розслідування інцидентів і контролю стану серверної інфраструктури. Водночас з’ясовано, що ефективність журналювання залежить не лише від факту збереження подій, а й від їх структурованості, наявності часових міток, ідентифікаторів джерел, рівнів серйозності та інших атрибутів, необхідних для подальшої машинної обробки.

Також у першому розділі розглянуто основні підходи до централізованого збирання, зберігання та обробки журналів подій. Визначено, що централізоване журналювання дає змогу подолати проблему розрізненості журналів у розподілених серверних середовищах і забезпечує єдину точку для пошуку, аналізу, пов’язування та моніторингу подій. Встановлено, що така система повинна підтримувати приймання даних із різних джерел, нормалізацію форматів, буферизацію, захищене передавання, керування строками зберігання, індексацію, аудит доступу та механізми сповіщення. Окрему увагу приділено питанням надійності, масштабованості та безпеки, оскільки втрата, спотворення або несанкціонований доступ до журналів можуть суттєво знизити їхню практичну й доказову цінність.

Крім того, здійснено огляд існуючих програмних рішень для моніторингу та аналізу журналів, зокрема Elastic Stack, Grafana Loki, Graylog, Wazuh, OpenSearch і Splunk. У результаті порівняння визначено, що кожне з цих рішень має власну архітектурну спрямованість. Одні системи краще підходять для повнотекстового пошуку та аналізу за різними полями, інші орієнтовані на зберігання великих потоків журналів з нижчими витратами, безпековий моніторинг або корпоративний SIEM-аналіз. Це дало змогу зробити висновок, що вибір підходу до побудови системи централізованого збирання журналів має ґрунтуватися на характері джерел даних, очікуваному обсязі подій, вимогах до пошуку, безпеки, зберігання та подальшої обробки.

Отже, у результаті аналізу встановлено, що для розроблення серверної системи централізованого збирання журналів подій недостатньо реалізувати лише приймання та збереження записів. Необхідно спроєктувати цілісну систему, яка забезпечує структуроване надходження подій, їх нормалізацію, надійне зберігання, фільтрацію, пошук, статистичну обробку, аудит і можливість подальшого розширення. Саме ці положення стали теоретичною та методологічною основою для подальшого проєктування архітектури, моделі даних і програмної реалізації серверної системи централізованого збирання журналів подій.

РОЗДІЛ 2. РЕАЛІЗАЦІЯ СЕРВЕРНОЇ СИСТЕМИ ЦЕНТРАЛІЗОВАНОГО ЗБОРУ ЖУРНАЛІВ ПОДІЙ

2.1 Проєктування архітектури серверної системи централізованого збору журналів подій

Для реалізації серверної системи централізованого збору журналів подій було спроєктовано архітектуру, яка забезпечує приймання журналів від декількох сервісів, їхню валідацію, обробку, збереження, індексацію за часом, формування статистики та можливість подальшого перегляду. Основна ідея системи полягає в тому, що різні сервіси не зберігають журнали подій окремо у власних локальних файлах, а передають їх до єдиного центрального сервера через HTTP API [25]. Такий підхід спрощує аналіз помилок, моніторинг роботи сервісів і пошук подій за рівнем критичності, назвою сервісу або часовим проміжком.

У спроєктованій системі джерелами журналів подій виступають демонстраційні сервіси auth-service, billing-service та notification-service. Кожен із цих сервісів імітує роботу окремого програмного модуля та формує події різного рівня важливості: INFO, WARNING, ERROR і CRITICAL. Сформовані журнали передаються до центрального сервера за допомогою HTTP-запиту POST /logs. Загальну архітектуру серверної системи централізованого збору журналів подій наведено на рисунку 2.1.

Архітектура системи має багаторівневу структуру. На першому рівні розташовані сервіси-джерела журналів подій. Вони відповідають за формування повідомлень про події, які виникають під час роботи програмної системи. Наприклад, auth-service може формувати журнали, пов’язані з авторизацією користувачів, billing-service – з оплатами або виставленням рахунків, а notification-service – з надсиланням повідомлень. Кожен журнал

містить рівень критичності, назву сервісу, текст повідомлення, хост, додаткові метадані та, за потреби, ідентифікатор проходження запиту.

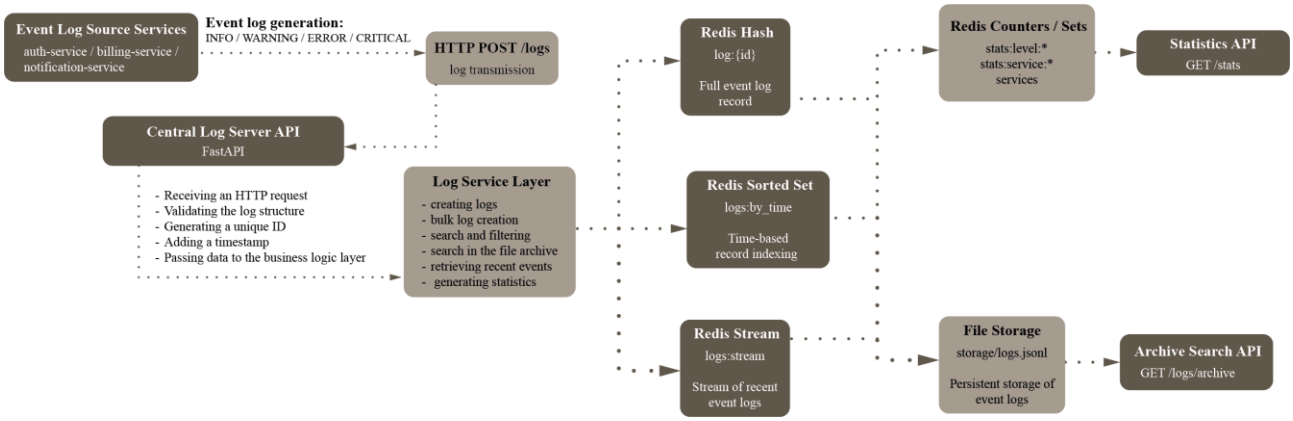


Рисунок 2.1 – Архітектура серверної системи централізованого збору журналів подій

Передача журналів до центральної системи виконується через HTTP-запит POST /logs. Такий спосіб взаємодії є зручним, оскільки REST API може використовуватися різними клієнтськими сервісами незалежно від їхньої внутрішньої реалізації. Сервісу-джерелу достатньо сформувати JSON-об’єкт [26] відповідної структури та надіслати його на центральний сервер. Це робить систему гнучкою та придатною для подальшого розширення, оскільки до неї можна підключати нові сервіси без зміни основної логіки центрального сервера.

Центральним компонентом архітектури є Central Log Server API, реалізований за допомогою FastAPI [27]. Цей рівень відповідає за приймання HTTP-запитів, перевірку коректності переданих даних, формування унікального ідентифікатора журналу та додавання часової мітки. Валідація структури журналу дозволяє відхиляти некоректні запити ще на рівні API, що підвищує надійність системи та запобігає збереженню неповних або помилкових даних.

Після проходження рівня API журнал передається на рівень бізнес-логіки – Log Service Layer. Цей рівень виконує основні операції з журналами подій: створення одного запису, масове створення записів, пошук, фільтрацію,

отримання останніх подій, пошук у файловому архіві та формування статистичних показників. Виділення окремого сервісного шару дозволяє відокремити логіку обробки журналів від маршрутизації HTTP-запитів. Завдяки цьому структура програмного коду стає зрозумілішою, а окремі компоненти системи легше підтримувати та змінювати.

Для зберігання журналів подій у системі використовується Redis [28]. Він виконує роль швидкого сховища даних і забезпечує ефективний доступ до журналів. Повний запис кожної події зберігається у структурі Redis Hash за ключем log:{id}. Такий підхід дозволяє швидко отримати повну інформацію про конкретний журнал за його унікальним ідентифікатором.

Для пошуку журналів за часом використовується структура Redis Sorted Set з ключем logs:by_time. У цій структурі ідентифікатори журналів зберігаються разом із числовим значенням часу. Це дає змогу швидко отримувати журнали за певний період, а також виводити останні записи у зворотному хронологічному порядку. Такий механізм є важливим для системи централізованого журналювання, оскільки аналіз подій найчастіше виконується саме за часовою послідовністю.

Окремо в архітектурі використовується Redis Stream з ключем logs:stream. Цей компонент призначений для зберігання потоку останніх подій. Його використання дає змогу швидко отримувати останні журнали без повного перегляду всього набору даних. Це корисно для побудови панелі моніторингу або перегляду актуального стану системи [29].

Для формування статистики застосовуються лічильники та структури Redis, які зберігають загальну кількість журналів, кількість подій за рівнями критичності та кількість подій за окремими сервісами. На основі цих даних реалізується API статистики GET /stats. За допомогою цього запиту можна отримати узагальнену інформацію про роботу системи, наприклад кількість помилок, попереджень або критичних подій.

Крім зберігання в Redis, у системі передбачено додаткове файлове сховище storage/logs.jsonl. Кожен журнал дублюється у файл у форматі JSON

Lines, де один рядок відповідає одному журналу події. Це забезпечує додаткове постійне збереження даних і дозволяє переглядати журнали навіть окремо від Redis. Для роботи з файловим архівом передбачено окремий API-запит GET /logs/archive, який дає змогу виконувати пошук журналів у файловому сховищі.

2.2 Реалізація REST API для приймання та перегляду журналів подій

Для взаємодії між сервісами-джерелами журналів подій та центральним сервером у розробленій системі реалізовано REST API. Воно забезпечує приймання журналів подій, масове додавання записів, перегляд збережених журналів, фільтрацію за рівнем критичності, назвою сервісу та часовим проміжком, а також отримання окремого журналу за його унікальним ідентифікатором.

REST API реалізовано за допомогою фреймворку FastAPI. Використання цього фреймворку дозволяє швидко створювати HTTP-ендпоінти, автоматично виконувати валідацію вхідних даних на основі Pydantic-схем, формувати документацію API та повертати відповіді у форматі JSON. У розробленій системі API виступає проміжним рівнем між клієнтськими сервісами та сервісним шаром LogService, який відповідає за основну логіку обробки журналів.

Ініціалізація FastAPI-додатку виконується у файлі app/main.py. У цьому модулі створюється екземпляр застосунку, задаються назва, версія, опис системи, підключається middleware для CORS та реєструються маршрути API. Фрагмент створення FastAPI-додатку наведено на рисунку 2.2.

```python
def create_app() -> FastAPI:
    settings = get_settings()

    app = FastAPI(
        title=settings.app_name,
        version=settings.app_version,
        description="Centralized Redis-based event log server with file archiving.",
    )

    app.add_middleware(
        CORSMiddleware,
        allow_origins=["*"],
        allow_credentials=True,
        allow_methods=["*"],
        allow_headers=["*"],
    )

    app.include_router(router)
    app.include_router(router, prefix="/api")
    app.include_router(router, prefix="/api/v1")

    return app
```

Рисунок 2.2 – Фрагмент коду створення FastAPI-додатку та підключення маршрутів API

У наведеному фрагменті видно, що маршрути API підключаються одразу в декількох варіантах: без префікса, з префіксом /api та з префіксом /api/v1. Це забезпечує зручність використання системи та можливість подальшого розмежування різних версій API. Основним варіантом для роботи із системою є шлях /api/v1, наприклад /api/v1/logs.

Для опису структури вхідних і вихідних даних у системі використовуються Pydantic-схеми [30]. Вони визначають, які поля повинен містити журнал події, які рівні журналювання допускаються та який формат відповіді повертає API. Основна схема створення журналу наведена на рисунку 2.3.

```python
class LogLevel(str, Enum):
    DEBUG = "DEBUG"
    INFO = "INFO"
    WARNING = "WARNING"
    ERROR = "ERROR"
    CRITICAL = "CRITICAL"


class LogCreate(BaseModel):
    level: LogLevel
    service: str = Field(min_length=1, max_length=100)
    message: str = Field(min_length=1, max_length=1000)
    host: str | None = Field(default=None, max_length=100)
    trace_id: str | None = Field(default=None, max_length=100)
    metadata: dict[str, Any] = Field(default_factory=dict)
```

Рисунок 2.3 – Фрагмент коду схеми журналу події

Схема LogCreate визначає структуру даних, які надсилаються сервісом-джерелом до центрального сервера. Обов'язковими полями є level, service та message. Поле level обмежене переліком допустимих значень: DEBUG, INFO, WARNING, ERROR та CRITICAL. Це дає змогу уніфікувати рівні журналювання в усій системі. Поля host, trace_id і metadata є додатковими та використовуються для збереження розширеної інформації про подію.

Для створення одного журналу події реалізовано API-маршрут для обробки HTTP-запитів [31] (HTTP-endpoint) POST /logs. Він приймає JSON-об'єкт, автоматично перевіряє його відповідність схемі LogCreate та передає дані на рівень бізнес-логіки. Фрагмент реалізації HTTP-endpoint додавання журналу наведено на рисунку 2.4.

```
@router.post( path: "/logs", response_model=LogResponse, status_code=status.HTTP_201_CREATED)
def create_log(payload: LogCreate, service: LogService = Depends(get_log_service)) -> LogResponse:
    return service.create_log(payload)
```

Рисунок 2.4 – Фрагмент коду HTTP-endpoint для створення журналу події

У цьому фрагменті HTTP-endpoint приймає об'єкт payload типу LogCreate. Якщо структура запиту є правильною, дані передаються до методу create_log() сервісу LogService. У разі успішного створення журналу API повертає відповідь зі статусом 201 Created. Це відповідає стандартній логіці REST API, де створення нового ресурсу супроводжується відповідним HTTP-статусом.

Крім створення одного журналу, у системі передбачено можливість масового додавання записів. Для цього реалізовано HTTP-endpoint POST /logs/batch, який приймає список журналів і повертає кількість доданих записів та їхні ідентифікатори. Фрагмент реалізації масового додавання журналів наведено на рисунку 2.5.

```python
@router.post( path: "/logs/batch", response_model=BatchLogResponse, status_code=status.HTTP_201_CREATED)
def create_logs_batch(payload: BatchLogCreate, service: LogService = Depends(get_log_service)) -> BatchLogResponse:
    created = service.create_many(payload.logs)
    return BatchLogResponse(inserted=len(created), ids=[item.id for item in created])
```

Рисунок 2.5 – Фрагмент коду HTTP-endpoint для масового створення журналів подій

Масове створення журналів є корисним у випадках, коли сервіс-джерело формує декілька подій і передає їх до центрального сервера одним запитом. Це зменшує кількість HTTP-звернень та спрощує передавання групи журналів у межах одного запиту.

Для перегляду журналів реалізовано HTTP-endpoint GET /logs. Він дозволяє отримувати список журналів із можливістю фільтрації за рівнем критичності, сервісом, початковою та кінцевою датою, а також обмеженням кількості результатів. Фрагмент реалізації цього HTTP-endpoint наведено на рисунку 2.6.

```python
@router.get( path: "/logs", response_model=LogListResponse)
def list_logs(
    level: LogLevel | None = None,
    service_name: str | None = Query(default=None, alias="service"),
    date_from: str | None = Query(default=None, description="For example: 2026-06-12T10:00:00Z"),
    date_to: str | None = Query(default=None, description="For example: 2026-06-12T23:59:59Z"),
    limit: int = Query(default=50, ge=1, le=500),
    log_service: LogService = Depends(get_log_service),
) -> LogListResponse:
    items = log_service.search_logs(
        level=level.value if level else None,
        service=service_name,
        date_from=date_from,
        date_to=date_to,
        limit=limit,
    )
    return LogListResponse(total=len(items), items=items)
```

Рисунок 2.6 – Фрагмент коду HTTP-endpoint для перегляду та фільтрації журналів подій

У цьому HTTP-endpoint параметр level використовується для фільтрації журналів за рівнем критичності, параметр service – для вибірки журналів

конкретного сервісу, a date_from і date_to – для пошуку подій у певному часовому проміжку. Параметр limit обмежує кількість записів, що повертаються у відповіді. Це дозволяє уникнути надмірного навантаження при великій кількості журналів.

Отримання конкретного журналу події реалізовано через HTTP-endpoint GET /logs/{log_id}. Він приймає унікальний ідентифікатор журналу та повертає повний запис події. Якщо журнал із таким ідентифікатором не знайдено, API повертає помилку 404 Not Found. Фрагмент реалізації цього HTTP-endpoint наведено на рисунку 2.7.

```
@router.get( path: "/logs/{log_id}", response_model=LogResponse)
def get_log(log_id: str, service: LogService = Depends(get_log_service)) -> LogResponse:
    item = service.get_log(log_id)
    if item is None:
        raise HTTPException(status_code=404, detail="Log event not found")
    return item
```

Рисунок 2.7 – Фрагмент коду HTTP-endpoint для отримання журналу події за ідентифікатором

Такий HTTP-endpoint потрібний у випадках, коли після перегляду загального списку журналів необхідно отримати детальну інформацію про конкретну подію. Наприклад, користувач може знайти помилку у списку подій, після чого відкрити її повний запис за ідентифікатором.

Окрім основних операцій із журналами, у REST API реалізовано службові HTTP-endpoint для перевірки стану системи, отримання останніх подій і перегляду статистики. Їх використання дозволяє не лише зберігати журнали, а й контролювати працездатність сервера та швидко аналізувати загальний стан системи. Основні маршрути REST API наведено в таблиці 2.1.

Реалізоване REST API забезпечує повний набір базових операцій для роботи із журналами подій. Воно приймає журнали від клієнтських сервісів, виконує валідацію структури даних, передає записи на рівень бізнес-логіки, дозволяє переглядати збережені журнали, застосовувати фільтри та отримувати окремі записи за ідентифікатором.

Таблиця 2.1 – Основні маршрути REST API

Метод	**Маршрут**	**Призначення**
GET	/api/v1/health	Перевірка стану сервера та підключення до Redis
POST	/api/v1/logs	Додавання одного журналу події
POST	/api/v1/logs/batch	Масове додавання журналів подій
GET	/api/v1/logs	Перегляд журналів із фільтрацією
GET	/api/v1/logs/{log_id}	Отримання конкретного журналу за ідентифікатором
GET	/api/v1/stream/latest	Отримання останніх журналів подій
GET	/api/v1/stats	Отримання статистики журналів
GET	/api/v1/logs/archive	Перегляд журналів із файлового архіву

Завдяки використанню FastAPI структура API є зрозумілою, з можливістю розширення та зручною для подальшого розвитку серверної системи централізованого збору журналів подій.

2.3 Реалізація механізмів зберігання, індексації та обробки журналів подій у Redis

Для забезпечення швидкого зберігання, пошуку та обробки журналів подій у розробленій серверній системі використовується Redis. У межах реалізації Redis виконує роль основного оперативного сховища, у якому зберігаються повні записи журналів, часові індекси, потік останніх подій та статистичні дані. Такий підхід дозволяє швидко отримувати журнали за унікальним ідентифікатором, виконувати вибірку за часовим проміжком, переглядати останні події та формувати узагальнену статистику роботи системи.

Підключення до Redis реалізовано в окремому модулі app/db.py. Для цього використовується функція get_redis(), яка отримує параметри конфігурації системи та створює клієнт Redis. Фрагмент програмного коду підключення до Redis наведено на рисунку 2.8.

```python
from redis import Redis

from app.config import get_settings


def get_redis() -> Redis:
    settings = get_settings()
    return Redis.from_url(settings.effective_redis_url, decode_responses=True)
```

Рисунок 2.8 – Фрагмент коду підключення до Redis

У наведеному фрагменті функція get_redis() створює підключення до Redis на основі значення effective_redis_url. Параметр decode_responses=True використовується для того, щоб дані, які повертаються з Redis, автоматично декодувалися у рядковий формат. Це спрощує подальшу обробку журналів подій у сервісному шарі системи.

Основна логіка роботи з журналами подій реалізована у класі LogService. У цьому класі визначено набір ключів Redis, які використовуються для зберігання журналів, індексації за часом, ведення потоку останніх подій та формування статистики. Фрагмент визначення основних Redis-ключів наведено на рисунку 2.9.

```python
class LogService:
    LOG_KEY_PREFIX = "log"
    LOG_TIME_INDEX = "logs:by_time"
    LOG_STREAM = "logs:stream"
    STATS_TOTAL = "stats:total"
    STATS_LEVEL = "stats:level"
    STATS_SERVICE = "stats:service"
```

Рисунок 2.9 – Фрагмент коду визначення основних Redis-ключів системи

Ключ LOG_KEY_PREFIX використовується для формування ключів повних записів журналів у форматі log:{id}. Ключ LOG_TIME_INDEX відповідає за часовий індекс журналів, LOG_STREAM використовується для потоку останніх подій, а ключі STATS_TOTAL, STATS_LEVEL та STATS_SERVICE призначені для збереження статистичних показників.

Під час створення нового журналу події система формує часову мітку, унікальний ідентифікатор та структуру запису. Кожен журнал містить ідентифікатор, час створення, рівень критичності, назву сервісу, повідомлення, хост, ідентифікатор трасування та додаткові метадані. Фрагмент формування журналу події наведено на рисунку 2.10.

```
def create_log(self, payload: LogCreate) -> LogResponse:
    now = datetime.now(timezone.utc)
    log_id = str(uuid.uuid4())

    item = {
        "id": log_id,
        "timestamp": now.isoformat(),
        "level": payload.level.value,
        "service": payload.service,
        "message": payload.message,
        "host": payload.host,
        "trace_id": payload.trace_id,
        "metadata": payload.metadata,
    }
```

Рисунок 2.10 – Фрагмент коду формування нового журналу події

У цьому фрагменті для кожного журналу створюється унікальний ідентифікатор за допомогою uuid.uuid4(). Час створення журналу фіксується у форматі UTC, що дозволяє уніфікувати часові дані незалежно від середовища, у якому працюють сервіси-джерела. Перед записом у Redis журнал події перетворюється у формат, придатний для збереження в Redis Hash. Оскільки Redis зберігає значення у вигляді рядків, складні структури, зокрема поле metadata, перетворюються на JSON-рядок. Фрагмент серіалізації журналу наведено на рисунку 2.11.

```
@staticmethod
def _serialize_for_redis(item: dict[str, Any]) -> dict[str, str]:
    return {
        "id": str(item.get( key: "id",  default: "")),
        "timestamp": str(item.get( key: "timestamp",  default: "")),
        "level": str(item.get( key: "level",  default: "")),
        "service": str(item.get( key: "service",  default: "")),
        "message": str(item.get( key: "message",  default: "")),
        "host": "" if item.get("host") is None else str(item.get("host")),
        "trace_id": "" if item.get("trace_id") is None else str(item.get("trace_id")),
        "metadata": json.dumps(item.get("metadata") or {}, ensure_ascii=False),
    }
```

Рисунок 2.11 – Фрагмент коду перетворення журналу для збереження в Redis

Завдяки такому перетворенню всі поля журналу мають рядковий формат, що забезпечує коректний запис даних у Redis. Для поля metadata використовується параметр ensure_ascii=False, що дозволяє зберігати текстові значення з українськими символами без їхнього примусового перетворення в escape-послідовності.

Основний процес збереження журналу в Redis реалізовано за допомогою механізму pipeline. Це дозволяє виконати декілька Redis-команд одним логічним блоком. У межах однієї операції система зберігає повний журнал у Redis Hash, додає ідентифікатор журналу до часового індексу Redis Sorted Set, записує подію до Redis Stream та оновлює статистичні лічильники. Фрагмент коду збереження журналу в Redis наведено на рисунку 2.12.

```
key = self._log_key(log_id)
redis_item = self._serialize_for_redis(item)

pipe = self.redis.pipeline()
pipe.hset(key, mapping=redis_item)
pipe.zadd(self.LOG_TIME_INDEX, mapping: {log_id: now.timestamp()})
pipe.xadd(self.LOG_STREAM, redis_item, maxlen=1000, approximate=True)
pipe.incr(self.STATS_TOTAL)
pipe.hincrby(self.STATS_LEVEL, payload.level.value, amount: 1)
pipe.hincrby(self.STATS_SERVICE, payload.service, amount: 1)
pipe.execute()
```

Рисунок 2.12 – Фрагмент коду збереження журналу, індексації та оновлення статистики в Redis

У наведеному фрагменті команда hset() записує повний журнал події у структуру Redis Hash. Ключ журналу формується у вигляді log:{id}, де {id} – унікальний ідентифікатор події. Команда zadd() додає журнал до структури Redis Sorted Set, де значенням сортування є час створення журналу. Це дає можливість виконувати швидкий пошук подій за часовим проміжком. Команда xadd() додає журнал до Redis Stream, який використовується для отримання останніх подій. Команди incr() та hincrby() оновлюють статистику загальної

кількості журналів, кількості журналів за рівнем критичності та кількості журналів за сервісом.

Для формування ключа конкретного журналу використовується допоміжний метод _log_key(). Він додає до унікального ідентифікатора журналу префікс log, завдяки чому всі записи журналів у Redis мають єдину структуру іменування. Фрагмент цього методу наведено на рисунку 2.13.

```
def _log_key(self, log_id: str) -> str:
    return f"{self.LOG_KEY_PREFIX}:{log_id}"
```

Рисунок 2.13 – Фрагмент коду формування ключа журналу події в Redis

Отримання окремого журналу за ідентифікатором виконується шляхом звернення до Redis Hash за відповідним ключем. Якщо журнал не знайдено, метод повертає значення None. Фрагмент коду отримання журналу за ідентифікатором наведено на рисунку 2.14.

```
def get_log(self, log_id: str) -> LogResponse | None:
    raw = self.redis.hgetall(self._log_key(log_id))
    if not raw:
        return None
    return self._to_response(self._deserialize_from_redis(raw))
```

Рисунок 2.14 – Фрагмент коду отримання журналу події з Redis Hash

Після отримання даних із Redis виконується зворотне перетворення. Поле metadata, яке зберігалося у вигляді JSON-рядка, перетворюється у словник Python. Фрагмент зворотного перетворення журналу наведено на рисунку 2.15.

```python
    @staticmethod
    def _deserialize_from_redis(raw: dict[str, str]) -> dict[str, Any]:
        metadata_raw = raw.get("metadata") or "{}"
        try:
            metadata = json.loads(metadata_raw)
        except json.JSONDecodeError:
            metadata = {}

        return {
            "id": raw.get("id"),
            "timestamp": raw.get("timestamp"),
            "level": raw.get("level"),
            "service": raw.get("service"),
            "message": raw.get("message"),
            "host": raw.get("host") or None,
            "trace_id": raw.get("trace_id") or None,
            "metadata": metadata,
        }
```

Рисунок 2.15 – Фрагмент коду зворотного перетворення журналу події з Redis

Пошук журналів за часовим проміжком реалізовано на основі Redis Sorted Set. Для цього в системі використовується ключ logs:by_time, у якому ідентифікатори журналів зберігаються разом із часовою міткою. Спочатку вхідні параметри date_from і date_to перетворюються на числові значення часу, після чого виконується вибірка ідентифікаторів журналів за допомогою zrevrangebyscore(). Фрагмент пошуку журналів наведено на рисунку 2.16.

```python
def search_logs(
    self,
    level: str | None = None,
    service: str | None = None,
    date_from: str | None = None,
    date_to: str | None = None,
    limit: int = 50,
) -> list[LogResponse]:
    min_score = self._parse_score(date_from, default="-inf")
    max_score = self._parse_score(date_to, default="+inf")

    ids = self.redis.zrevrangebyscore(
        self.LOG_TIME_INDEX,
        max=max_score,
        min=min_score,
        start=0,
        num=limit * 5,
    )
```

Рисунок 2.16 – Фрагмент коду пошуку журналів за часовим індексом Redis Sorted Set

Метод zrevrangebyscore() повертає ідентифікатори журналів у зворотному хронологічному порядку. Це означає, що найновіші журнали будуть отримані першими. Для кожного знайденого ідентифікатора система додатково отримує повний запис із Redis Hash, після чого застосовує фільтрацію за рівнем критичності та назвою сервісу. Фрагмент фільтрації журналів наведено на рисунку 2.17.

```
        items: list[LogResponse] = []
        for log_id in ids:
            raw = self.redis.hgetall(self._log_key(log_id))
            if not raw:
                continue

            item = self._deserialize_from_redis(raw)
            if level and item.get("level") != level:
                continue
            if service and item.get("service") != service:
                continue

            items.append(self._to_response(item))
            if len(items) >= limit:
                break

        return items
```

Рисунок 2.17 – Фрагмент коду фільтрації журналів за рівнем критичності та сервісом

Такий підхід дозволяє поєднати часову індексацію з додатковими умовами фільтрації. Спочатку Redis швидко повертає список журналів за періодом часу, а потім у сервісному шарі виконується уточнення результатів за рівнем журналу та назвою сервісу.

Для коректної роботи пошуку за датами реалізовано допоміжний метод _parse_score(). Він перетворює дату у форматі ISO 8601 у числову часову мітку, яка може використовуватися Redis Sorted Set як значення для порівняння. Фрагмент цього методу наведено на рисунку 2.18.

```python
@staticmethod
def _parse_score(value: str | None, default: str) -> float | int | str:
    if not value:
        return default
    normalized = value.replace( old: "Z",  new: "+00:00")
    parsed = datetime.fromisoformat(normalized)
    if parsed.tzinfo is None:
        parsed = parsed.replace(tzinfo=timezone.utc)
    return parsed.timestamp()
```

Рисунок 2.18 – Фрагмент коду перетворення дати у часову мітку для Redis Sorted Set

Окремим механізмом обробки журналів є отримання останніх подій із Redis Stream. Цей механізм використовується тоді, коли потрібно швидко переглянути не всі журнали, а лише останні події, що були передані до системи. Фрагмент отримання останніх подій із Redis Stream наведено на рисунку 2.19.

```python
def latest_stream_events(self, count: int = 10) -> list[LogResponse]:
    stream_items = self.redis.xrevrange(self.LOG_STREAM, count=count)
    result: list[LogResponse] = []

    for _, raw in stream_items:
        result.append(self._to_response(self._deserialize_from_redis(raw)))

    return result
```

Рисунок 2.19 – Фрагмент коду отримання останніх журналів із Redis Stream

Метод xrevrange() повертає останні записи потоку у зворотному порядку. Це дозволяє використовувати Redis Stream як джерело актуальних подій для перегляду останніх журналів або подальшого розширення системи у напрямі моніторингу в режимі реального часу.

Крім зберігання та пошуку журналів, система формує статистику. Для цього використовуються окремі ключі Redis. Загальна кількість журналів зберігається у ключі stats:total, кількість журналів за рівнем критичності – у хеші stats:level, а кількість журналів за сервісами – у хеші stats:service. Фрагмент отримання статистики наведено на рисунку 2.20.

```
def stats(self) -> dict[str, Any]:
    return {
        "total": int(self.redis.get(self.STATS_TOTAL) or 0),
        "by_level": {k: int(v) for k, v in self.redis.hgetall(self.STATS_LEVEL).items()},
        "by_service": {k: int(v) for k, v in self.redis.hgetall(self.STATS_SERVICE).items()},
    }
```

Рисунок 2.20 – Фрагмент коду отримання статистичних даних із Redis

Отримані статистичні дані використовуються API-запитом GET /stats. Завдяки цьому користувач системи може швидко отримати інформацію про загальну кількість журналів, розподіл подій за рівнем критичності та активність окремих сервісів. Це є важливим для моніторингу стану системи та швидкого виявлення проблемних компонентів.

Також у системі реалізовано можливість масового створення журналів. Для цього метод create_many() послідовно викликає метод create_log() для кожного елемента зі списку. Фрагмент коду масового створення журналів наведено на рисунку 2.21.

```
def create_many(self, logs: list[LogCreate]) -> list[LogResponse]:
    return [self.create_log(item) for item in logs]
```

Рисунок 2.21 – Фрагмент коду масового створення журналів подій

Масове створення журналів є корисним у випадках, коли сервіс-джерело накопичує декілька подій і передає їх до центральної системи одним запитом. При цьому кожен журнал проходить той самий процес обробки: формування ідентифікатора, додавання часової мітки, серіалізацію, запис у Redis Hash, індексацію в Redis Sorted Set, додавання в Redis Stream та оновлення статистики.

2.4 Реалізація додаткового файлового архіву журналів подій

Окрім основного зберігання журналів подій у Redis, у розробленій серверній системі реалізовано додатковий файловий архів. Його призначення полягає у дублюванні кожного прийнятого журналу у файл storage/logs.jsonl. Це

дозволяє зберігати копії подій у простому текстовому форматі та переглядати їх навіть окремо від Redis.

Для реалізації файлового архіву було обрано формат JSON Lines. Особливість цього формату полягає в тому, що кожен рядок файлу є окремим JSON-об’єктом. Такий підхід є зручним для журналів подій, оскільки нові записи можна додавати в кінець файлу без повного перезапису всього архіву.

Шлях до файлового архіву задається у конфігурації системи. За замовчуванням журнали зберігаються у файлі storage/logs.jsonl. Фрагмент конфігурації шляху до файлового архіву наведено на рисунку 2.22.

```
class Settings(BaseSettings):
    app_name: str = "Central Log Server Redis"
    app_version: str = "1.0.0"

    redis_url: str | None = None
    redis_host: str = "127.0.0.1"
    redis_port: int = 6379
    redis_db: int = 0

    demo_mode: bool = True
    log_storage_path: str = "storage/logs.jsonl"
    log_stream_max_len: int = 10000
```

Рисунок 2.22 – Фрагмент коду конфігурації шляху до файлового архіву журналів

У наведеному фрагменті параметр log_storage_path визначає розташування файлу, у який будуть записуватися журнали подій. Завдяки винесенню цього параметра в конфігурацію шлях до архіву можна змінювати без зміни основної логіки програми.

Основна робота з файловим архівом реалізована в окремому класі JsonlLogStorage. Під час створення об’єкта цього класу система перевіряє наявність каталогу для зберігання журналів, створює його за потреби та ініціалізує файл архіву. Фрагмент реалізації створення файлового сховища та запису журналу наведено на рисунку 2.23.

```python
class JsonlLogStorage:
    def __init__(self, path: str) -> None:
        self.path = Path(path)
        self.path.parent.mkdir(parents=True, exist_ok=True)
        self.path.touch(exist_ok=True)

    def append(self, item: dict[str, Any]) -> None:
        with self.path.open( mode: "a", encoding="utf-8") as file:
            file.write(json.dumps(item, ensure_ascii=False) + "\n")
```

Рисунок 2.23 – Фрагмент коду створення файлового архіву та додавання журналу

Метод append() відкриває файл у режимі додавання та записує до нього один журнал події у форматі JSON. Параметр ensure_ascii=False використовується для коректного збереження тексту з українськими символами. Після кожного JSON-об’єкта додається символ нового рядка, що забезпечує відповідність формату JSON Lines. Файловий архів підключається на рівні сервісу обробки журналів. У класі LogService створюється об’єкт JsonlLogStorage, якому передається шлях до файлу з конфігурації. Після запису журналу в Redis цей самий запис додатково дублюється у файловий архів. Фрагмент інтеграції файлового архіву із сервісним шаром наведено на рисунку 2.24.

```python
class LogService:
    def __init__(self, redis_client: Redis) -> None:
        self.redis = redis_client
        self.settings = get_settings()
        self.file_storage = JsonlLogStorage(self.settings.log_storage_path)
    def create_log(self, payload: LogCreate) -> LogResponse:
        now = datetime.now(timezone.utc)
        log_id = str(uuid.uuid4())

        item = {
            "id": log_id,
            "timestamp": now.isoformat(),
            "level": payload.level.value,
            "service": payload.service,
            "message": payload.message,
            "host": payload.host,
            "trace_id": payload.trace_id,
            "metadata": payload.metadata,
        }
        self.file_storage.append(item)
        return self._to_response(item)
```

Рисунок 2.24 – Фрагмент коду дублювання журналу події у файловий архів

У цьому фрагменті видно, що файловий архів є частиною сервісного шару системи. Після формування структури журналу події запис передається до методу append(), який зберігає його у файлі storage/logs.jsonl. Таким чином, кожна подія, що надходить до центрального сервера, зберігається не лише в Redis, а й у додатковому файловому сховищі.

Для перегляду журналів із файлового архіву реалізовано метод читання записів за заданими умовами. Він дозволяє фільтрувати журнали за часовим проміжком, назвою сервісу та рівнем критичності. Фрагмент реалізації пошуку у файловому архіві наведено на рисунку 2.25.

```
def read_by_period(
    self,
    date_from: str | None = None,
    date_to: str | None = None,
    service: str | None = None,
    level: str | None = None,
    limit: int = 100,
) -> list[dict[str, Any]]:
    start = self._parse_date(date_from) if date_from else None
    end = self._parse_date(date_to) if date_to else None

    result: list[dict[str, Any]] = []
    for item in self.read_all():
        timestamp = self._parse_date(item.get("timestamp"))
        if timestamp is None:
            continue
        if start and timestamp < start:
            continue
        if end and timestamp > end:
            continue
        if service and item.get("service") != service:
            continue
        if level and item.get("level") != level:
            continue
        result.append(item)

    result.sort(key=lambda row :{get} : row.get("timestamp", ""), reverse=True)
    return result[:limit]
```

Рисунок 2.25 – Фрагмент коду пошуку журналів у файловому архіві

Метод read_by_period() спочатку перетворює вхідні дати у формат, придатний для порівняння, після чого послідовно переглядає всі записи файлового архіву. Для кожного журналу перевіряється час створення, назва сервісу та рівень критичності. Якщо запис відповідає заданим умовам, він

додається до результату. Наприкінці результати сортуються у зворотному хронологічному порядку, тобто найновіші журнали відображаються першими.

Щоб надати доступ до файлового архіву через REST API, у системі реалізовано окремий HTTP-endpoint GET /logs/archive. Він приймає параметри фільтрації, передає їх у сервісний шар та повертає знайдені журнали у стандартному форматі відповіді. Фрагмент реалізації API для перегляду файлового архіву наведено на рисунку 2.26.

```python
@router.get( path: "/logs/archive", response_model=LogListResponse)
def list_logs_from_file_storage(
    level: LogLevel | None = None,
    service_name: str | None = Query(default=None, alias="service"),
    date_from: str | None = Query(default=None, description="For example:2026-06-12T10:00:00Z"),
    date_to: str | None = Query(default=None, description="For example:2026-06-12T23:59:59Z"),
    limit: int = Query(default=100, ge=1, le=1000),
    log_service: LogService = Depends(get_log_service),
) -> LogListResponse:
    items = log_service.search_file_logs(
        level=level.value if level else None,
        service=service_name,
        date_from=date_from,
        date_to=date_to,
        limit=limit,
    )
    return LogListResponse(total=len(items), items=items)
```

Рисунок 2.26 – Фрагмент коду API для перегляду журналів із файлового архіву

Завдяки цьому HTTP-endpoint користувач може отримати журнали не з Redis, а безпосередньо з файлового архіву. Наприклад, можна переглянути всі помилки певного сервісу або вибрати журнали за конкретний часовий проміжок. Це підвищує наочність роботи системи та демонструє наявність додаткового рівня зберігання даних.

Фактичний вигляд одного запису у файлі storage/logs.jsonl має формат JSON-об’єкта. Кожен рядок містить окремий журнал події з усіма основними полями: ідентифікатором, часовою міткою, рівнем критичності, назвою сервісу, повідомленням, хостом, ідентифікатором трасування та метаданими. Приклад такого запису наведено на рисунку 2.27.

```
{"id": "c0ff4da8-493c-42ec-b2da-e5b3fc415daa",
  "timestamp": "2026-06-12T19:42:06.534493+00:00",
  "level": "ERROR",
  "service": "auth-service",
  "message": "Invalid password for user admin",
  "host": "api-01",
  "trace_id": "trace-demo-001",
  "metadata": {
    "user_id": 101,
    "ip": "192.168.1.10"
  }
}
```


Рисунок 2.27 – Приклад запису журналу події у файловому архіві

Додатковий файловий архів у розробленій системі виконує роль резервного та демонстраційного сховища журналів подій. Він забезпечує постійне збереження кожного журналу у форматі JSON Lines, дозволяє переглядати записи незалежно від Redis і підтримує пошук за основними параметрами. Така реалізація підвищує надійність системи та робить процес зберігання журналів більш прозорим для подальшого аналізу.

2.5 Реалізація збору статистики та отримання останніх журналів подій

У розробленій серверній системі централізованого збору журналів подій реалізовано окремі можливості для перегляду статистики та отримання останніх подій. Ці функції не відповідають за створення нових журналів, а використовуються для аналізу вже збережених даних. Завдяки цьому користувач може швидко отримати загальну кількість подій, розподіл журналів за рівнями критичності, статистику за сервісами, а також список останніх записів, які були передані до системи.

Для повернення статистичних даних у системі використовується схема StatsResponse. Вона описує структуру відповіді API та містить три основні поля: загальну кількість журналів, кількість подій за рівнями критичності та кількість подій за сервісами. Фрагмент коду цієї схеми наведено на рисунку 2.28.

```python
class StatsResponse(BaseModel):
    total: int
    by_level: dict[str, int]
    by_service: dict[str, int]
```

Рисунок 2.28 – Фрагмент коду схеми відповіді для статистики журналів

Поле total використовується для відображення загальної кількості журналів подій, які були прийняті системою. Поле by_level містить статистику за рівнями, наприклад INFO, WARNING, ERROR або CRITICAL. Поле by_service використовується для відображення кількості подій, отриманих від кожного сервісу-джерела.

Отримання статистичних даних реалізовано в сервісному шарі у методі stats(). Цей метод звертається до Redis, зчитує необхідні значення та перетворює їх у числовий формат. Фрагмент реалізації методу отримання статистики наведено на рисунку 2.29.

```python
def stats(self) -> dict[str, Any]:
    return {
        "total": int(self.redis.get(self.STATS_TOTAL) or 0),
        "by_level": {k: int(v) for k, v in self.redis.hgetall(self.STATS_LEVEL).items()},
        "by_service": {k: int(v) for k, v in self.redis.hgetall(self.STATS_SERVICE).items()},
    }
```

Рисунок 2.29 – Фрагмент коду отримання статистичних даних із Redis

У цьому методі значення загальної кількості журналів отримується з ключа STATS_TOTAL. Якщо значення відсутнє, використовується 0, що дозволяє коректно повертати статистику навіть тоді, коли в системі ще немає жодного журналу. Дані за рівнями критичності зчитуються з Redis Hash STATS_LEVEL, а дані за сервісами – з Redis Hash STATS_SERVICE. Оскільки Redis повертає значення у рядковому форматі, у методі виконується перетворення кожного значення в тип int.

Для доступу до статистики через HTTP-запит у REST API реалізовано HTTP-endpoint GET /stats. Він викликає метод stats() сервісного шару та

повертає результат у форматі StatsResponse. Фрагмент реалізації цього HTTP-endpoint наведено на рисунку 2.30.

```
@router.get( path: "/stats", response_model=StatsResponse)
def get_stats(service: LogService = Depends(get_log_service)) -> StatsResponse:
    return StatsResponse(**service.stats())
```

Рисунок 2.30 – Фрагмент коду HTTP-endpoint для отримання статистики журналів

Завдяки цьому HTTP-endpoint користувач може швидко отримати узагальнену інформацію про стан. Наприклад, можна визначити, скільки всього журналів було прийнято системою, скільки з них мають рівень ERROR або CRITICAL, а також які сервіси створюють найбільше записів.

Окрім статистики, у системі реалізовано отримання останніх подій. Для цього використовується метод latest_stream_events(), який звертається до Redis Stream і повертає задану кількість останніх журналів. Фрагмент реалізації цього методу наведено на рисунку 2.31.

```
def latest_stream_events(self, count: int = 10) -> list[LogResponse]:
    stream_items = self.redis.xrevrange(self.LOG_STREAM, count=count)
    result: list[LogResponse] = []

    for _, raw in stream_items:
        result.append(self._to_response(self._deserialize_from_redis(raw)))

    return result
```

Рисунок 2.31 – Фрагмент коду отримання останніх подій з Redis Stream

У наведеному методі використовується команда xrevrange(), яка отримує записи з потоку у зворотному порядку, тобто від найновіших до старіших. Параметр count визначає кількість журналів, які потрібно повернути. Після отримання даних кожен запис перетворюється з формату Redis у внутрішню структуру програми, а потім у відповідь типу LogResponse.

Для доступу до останніх журналів через API реалізовано HTTP-endpoint GET /stream/latest. Він приймає параметр count, викликає відповідний метод

сервісного шару та повертає список останніх подій. Фрагмент реалізації HTTP-endpoint наведено на рисунку 2.32.

```
@router.get( path: "/stream/latest", response_model=LogListResponse)
def latest_stream_events(
    count: int = Query(default=10, ge=1, le=100),
    service: LogService = Depends(get_log_service),
) -> LogListResponse:
    items = service.latest_stream_events(count=count)
    return LogListResponse(total=len(items), items=items)
```

Рисунок 2.32 – Фрагмент коду HTTP-endpoint для отримання останніх журналів подій

У цьому HTTP-endpoint параметр count має обмеження від 1 до 100. Таке обмеження потрібне для того, щоб користувач не міг випадково або навмисно запитати надто велику кількість записів одним HTTP-запитом. За замовчуванням система повертає 10 останніх подій.

У системі реалізовано два окремі механізми перегляду стану. Перший механізм забезпечує отримання статистики через HTTP-endpoint GET /stats, а другий – перегляд останніх подій через HTTP-endpoint GET /stream/latest. Статистика дає змогу оцінити загальний стан системи та активність сервісів, а отримання останніх подій дозволяє швидко переглянути найновіші журнали без повного пошуку по всьому сховищу.

2.6 Висновок до другого розділу

У другому розділі було розглянуто реалізацію серверної системи централізованого збору журналів подій. Було описано загальну архітектуру системи, яка передбачає приймання журналів від декількох сервісів-джерел, їхню валідацію, обробку, збереження, індексацію, формування статистики та можливість подальшого перегляду через REST API.

Було спроєктовано архітектуру системи на основі FastAPI-застосунку, сервісного шару LogService, сховища Redis та додаткового файлового архіву.

Джерелами журналів подій виступають окремі сервіси, зокрема auth-service, billing-service та notification-service, які передають події різних рівнів критичності до центрального сервера за допомогою HTTP-запитів. Такий підхід дозволяє зосередити журнали в одному місці та спростити їх подальший аналіз.

У системі реалізовано REST API для створення одного журналу, масового додавання записів, перегляду журналів із фільтрацією, отримання окремого запису за ідентифікатором, перегляду останніх подій, отримання статистики та пошуку у файловому архіві. Для перевірки структури вхідних даних використовуються Pydantic-схеми, що забезпечує коректність журналів ще на етапі приймання HTTP-запиту.

Основним оперативним сховищем системи є Redis. У ньому реалізовано збереження повних записів журналів у Redis Hash, індексацію за часом у Redis Sorted Set, зберігання останніх подій у Redis Stream та ведення статистичних лічильників. Завдяки цьому система може швидко отримувати журнали за ідентифікатором, виконувати пошук за часовим проміжком, виводити останні події та формувати узагальнену статистику за рівнями критичності й сервісами.

Окремо було реалізовано додатковий файловий архів у форматі JSON Lines. Кожен прийнятий журнал дублюється у файл storage/logs.jsonl, що забезпечує додаткове постійне збереження даних. Файловий архів дозволяє переглядати журнали незалежно від Redis та виконувати пошук за основними параметрами, зокрема датою, рівнем критичності та назвою сервісу.

Також у системі реалізовано механізми отримання статистики та останніх подій. HTTP-endpoint GET /stats дозволяє отримати загальну кількість журналів, розподіл подій за рівнями критичності та статистику за сервісами. HTTP-endpoint GET /stream/latest забезпечує швидкий перегляд найновіших подій без необхідності виконувати повний пошук у сховищі.

Отже, у другому розділі було реалізовано повноцінну серверну систему централізованого збору журналів подій. Розроблена система забезпечує приймання, валідацію, збереження, індексацію, фільтрацію, архівування та аналіз журналів. Використання FastAPI, Redis і файлового архіву дозволило

створити гнучке та розширюване рішення, яке може застосовуватися для моніторингу роботи сервісів, пошуку помилок і контролю стану програмної системи.

РОЗДІЛ 3. ТЕСТУВАННЯ РОБОТИ СЕРВЕРНОЇ СИСТЕМИ ЦЕНТРАЛІЗОВАНОГО ЗБОРУ ЖУРНАЛІВ ПОДІЙ

3.1 Підготовка та запуск тестового середовища

Для перевірки працездатності серверної системи централізованого збору журналів подій було підготовлено локальне тестове середовище. Тестування виконувалося на основі розробленого FastAPI-застосунку, Redis-сховища та додаткового файлового архіву журналів подій. Основною метою цього етапу було перевірити можливість запуску системи, доступність API-сервера та коректність підключення до Redis.

У тестовому середовищі використовувалися такі компоненти: серверний застосунок FastAPI, база Redis для зберігання журналів, Docker Compose для запуску сервісів, а також утиліта curl для перевірки доступності REST API. Згідно з конфігурацією проєкту, API-сервер запускається на порту 8000, а Redis - на порту 6379.

Перед запуском системи було перевірено структуру проєкту та наявність основних файлів, необхідних для тестування. До таких файлів належать Dockerfile, docker-compose.yml, .env, каталог app із програмною логікою, каталог services із демонстраційними сервісами та каталог scripts із демонстраційним сценарієм запитів. Результат перегляду структури проєкту наведено на рисунку 3.1.

```
central-log-server-redis % ls
Dockerfile              README.md               docker-compose.yml      scripts                 storage
Makefile                app                     requirements.txt        services                tests
```

Рисунок 3.1 – Перевірка структури проєкту перед запуском тестового середовища

Після перевірки структури проєкту було налаштовано файл `.env`, у якому задаються основні параметри запуску системи. У цьому файлі визначено назву застосунку, версію, адресу Redis, режим демонстрації та шлях до файлового архіву журналів. Приклад вмісту конфігураційного файлу наведено на рисунку 3.2.

```
APP_NAME=Central Log Server Redis
APP_VERSION=1.0.0

REDIS_URL=redis://redis:6379/0

DEMO_MODE=true
LOG_STORAGE_PATH=storage/logs.jsonl
LOG_STREAM_MAX_LEN=10000
```

Рисунок 3.2 – Конфігураційні параметри тестового середовища

Для запуску системи використовувався Docker Compose. У файлі docker-compose.yml передбачено два основні сервіси: redis та api. Сервіс redis запускає Redis-сервер, а сервіс api збирає та запускає FastAPI-застосунок. Для запуску тестового середовища було виконано команду, результат якої наведено на рисунку 3.3.

```
central-log-server-redis % docker compose up --build -d
[+] Building 0.5s (13/13) FINISHED
 => [internal] load local bake definitions                                                                                0.0s
 => => reading from stdin 601B                                                                                            0.0s
 => [internal] load build definition from Dockerfile                                                                      0.0s
 => => transferring dockerfile: 267B                                                                                      0.0s
 => [internal] load metadata for docker.io/library/python:3.12-slim                                                       0.0s
 => [internal] load .dockerignore                                                                                         0.0s
 => => transferring context: 2B                                                                                           0.0s
 => [1/6] FROM docker.io/library/python:3.12-slim@sha256:401f6e1a67dad31a1bd78e9ad22d0ee0a3b52154e6bd30e90be696bb6a3d7461 0.0s
 => => resolve docker.io/library/python:3.12-slim@sha256:401f6e1a67dad31a1bd78e9ad22d0ee0a3b52154e6bd30e90be696bb6a3d7461 0.0s
 => [internal] load build context                                                                                         0.1s
 => => transferring context: 320.90kB                                                                                     0.1s
 => CACHED [2/6] WORKDIR /app                                                                                             0.0s
 => CACHED [3/6] COPY requirements.txt .                                                                                  0.0s
 => CACHED [4/6] RUN pip install --no-cache-dir -r requirements.txt                                                       0.0s
 => CACHED [5/6] COPY . .                                                                                                 0.0s
 => CACHED [6/6] RUN mkdir -p storage                                                                                     0.0s
 => exporting to image                                                                                                    0.1s
 => => exporting layers                                                                                                   0.0s
 => => exporting manifest sha256:9e8bfcaa71763f5fb4c9f5a214808af4bd14f59a173103d8480277da9e645283                         0.0s
 => => exporting config sha256:e15da3de8b34a903247b5c728e1233b2a747b53896a91ac81acfb3585d2b98a9                           0.0s
 => => exporting attestation manifest sha256:d1091782294f981b94899b1d1b96e6e70a27d309759de5e390271b11ecc1b04a             0.0s
 => => exporting manifest list sha256:6e652df346ddf740749eaba94abb1a0ec5e5dc8790c4ac6555b58f5c5513e25c                    0.0s
 => => naming to docker.io/library/central-log-server-redis-api:latest                                                    0.0s
 => => unpacking to docker.io/library/central-log-server-redis-api:latest                                                 0.0s
 => resolving provenance for metadata file                                                                                0.0s
[+] Running 3/3
 ✔ central-log-server-redis-api  Built                                                                                    0.0s
 ✔ Container central-log-redis   Running                                                                                  0.0s
 ✔ Container central-log-api     Started                                                                                  0.7s
```

Рисунок 3.3 – Запуск тестового середовища за допомогою Docker Compose

Після запуску контейнерів було перевірено їхній стан. Для цього використано команду docker compose ps, яка показує список запущених сервісів, їхній статус і відкриті порти. Результат перевірки контейнерів наведено на рисунку 3.4.

```
central-log-server-redis % docker compose ps
NAME                IMAGE                          COMMAND                  SERVICE   CREATED          STATUS          PORTS
central-log-api     central-log-server-redis-api   "uvicorn app.main:ap…"   api       59 seconds ago   Up 58 seconds   0.0.0.0:8000->8000/tcp, [::]:8000->8000/tcp
central-log-redis   redis:7-alpine                 "docker-entrypoint.s…"   redis     11 hours ago     Up 5 minutes    0.0.0.0:6379->6379/tcp, [::]:6379->6379/tcp
```

Рисунок 3.4 – Перевірка стану контейнерів серверної системи

З рисунка 3.4 видно, що обидва контейнери перебувають у стані Up. Це означає, що Redis-сервер та FastAPI-застосунок були успішно запущені. API-сервер доступний на порту 8000, а Redis - на порту 6379.

Для додаткової перевірки запуску було виконано запит до службового HTTP-endpoint GET /api/v1/health. Цей запит дозволяє перевірити загальний стан API-сервера та наявність підключення до Redis. Результат виконання health-check наведено на рисунку 3.5.

```
central-log-server-redis % curl -s http://127.0.0.1:8000/api/v1/health | python3 -m json.tool
{
    "status": "ok",
    "app": "Central Log Server Redis",
    "version": "1.0.0",
    "redis": "ok"
}
```

Рисунок 3.5 – Перевірка працездатності API-сервера та підключення до Redis

Отримана відповідь свідчить про те, що серверний застосунок працює коректно. Поле status має значення ok, що означає нормальний стан системи. Поле redis також має значення ok, тобто API-сервер успішно підключився до Redis. Поля app і version підтверджують, що запущено саме розроблений застосунок Central Log Server Redis версії 1.0.0.

Також було перевірено доступність автоматично згенерованої документації FastAPI. Для цього в браузері було відкрито адресу http://127.0.0.1:8000/docs. Наявність сторінки Swagger UI підтверджує, що REST API доступне для перегляду, тестування та виконання запитів. Приклад адреси для відкриття документації наведено на рисунку 3.6.

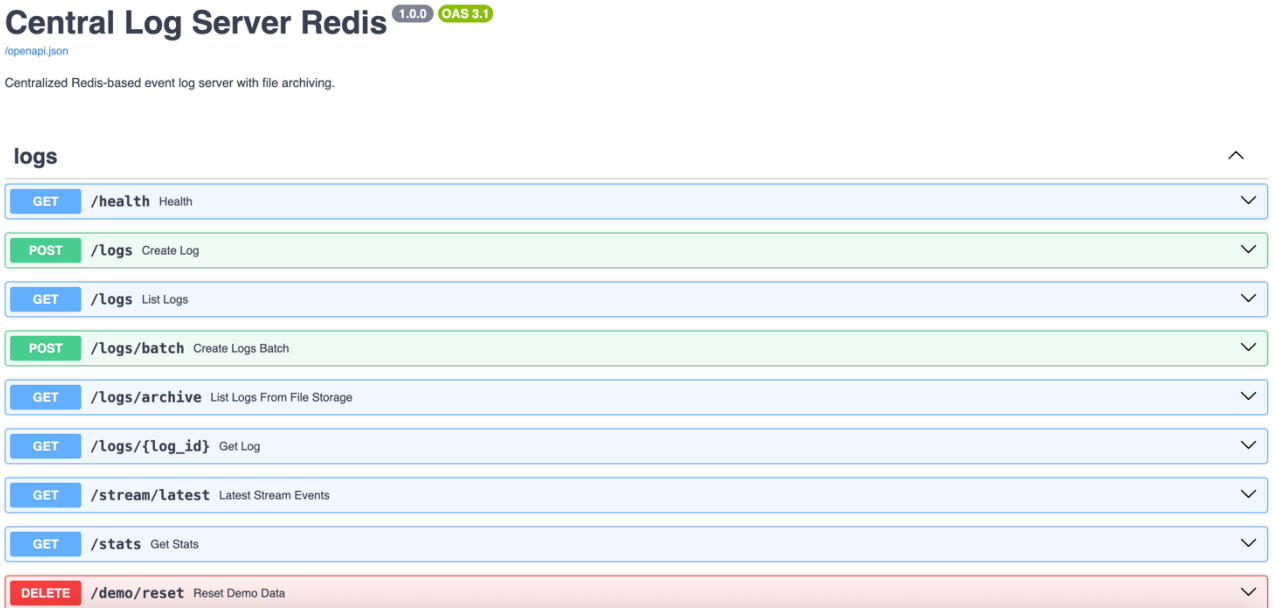


Рисунок 3.6 – Адреса Swagger UI для перегляду та тестування REST API

Для спрощення запуску в проєкті також передбачено команди в Makefile. Зокрема, команда make docker-up запускає систему через Docker Compose, команда make docker-down зупиняє контейнери, а команда make demo виконує демонстраційний сценарій запитів до API. Фрагмент команд для запуску тестового середовища наведено на рисунку 3.7.

```
run:
    uvicorn app.main:app --reload

test:
    pytest -q

docker-up:
    docker compose up --build

docker-down:
    docker compose down

demo:
    bash scripts/demo_requests.sh
```

Рисунок 3.7 – Команди Makefile для запуску та перевірки системи

На етапі підготовки тестового середовища було перевірено структуру проєкту, конфігураційні параметри, запуск контейнерів Docker, доступність API-сервера та підключення до Redis. Результат health-check підтвердив, що система перебуває у працездатному стані та готова до подальшого тестування приймання, зберігання, фільтрації й аналізу журналів подій.

3.2 Перевірка приймання та зберігання журналів подій

Після запуску тестового середовища було виконано перевірку приймання журналів подій через REST API. Для цього використано демонстраційний сценарій scripts/demo_requests.sh, який послідовно виконує службову перевірку системи, очищення демонстраційних даних, створення одного журналу події та масове створення декількох журналів.

На початку тестування було виконано перевірку стану системи за допомогою запиту до HTTP-endpoint /api/v1/health. Результат виконання запиту наведено на рисунку 3.8.

```
central-log-server-redis % bash scripts/demo_requests.sh
Health-check
{
    "status": "ok",
    "app": "Central Log Server Redis",
    "version": "1.0.0",
    "redis": "ok"
}
```

Рисунок 3.8 – Результат перевірки стану серверної системи

З рисунка 3.8 видно, що серверна система перебуває у працездатному стані. Поле status має значення ok, а поле redis також має значення ok, що підтверджує успішне підключення FastAPI-застосунку до Redis.

Перед створенням нових журналів було виконано очищення демонстраційних даних. Це дозволило проводити тестування в однакових

умовах і переконатися, що подальші результати сформовані саме під час поточного запуску сценарію. Результат очищення даних наведено на рисунку 3.9.

```
Reset demo data
HTTP 204
```

Рисунок 3.9 – Результат очищення демонстраційних даних

Код відповіді HTTP 204 означає, що операція очищення виконана успішно, але тіло відповіді відсутнє. Така поведінка є коректною для операцій, які не повертають додаткових даних після виконання.

Далі було перевірено створення одного журналу події. Для цього сценарій надіслав POST-запит до HTTP-endpoint /api/v1/logs із даними про помилку автентифікації в сервісі auth-service. Результат створення одного журналу наведено на рисунку 3.10.

```
Create one log event
{
    "id": "03435a52-4864-4020-b7f1-c95b43de3800",
    "timestamp": "2026-06-12T23:44:24.747154Z",
    "level": "ERROR",
    "service": "auth-service",
    "message": "Invalid password for user admin",
    "host": "api-01",
    "trace_id": "trace-demo-001",
    "metadata": {
        "user_id": 101,
        "ip": "192.168.1.10"
    }
}
```

Рисунок 3.10 – Результат створення одного журналу події

Отримана відповідь підтверджує, що сервер прийняв журнал події, сформував для нього унікальний ідентифікатор id, додав часову мітку timestamp та повернув повний об’єкт створеного журналу. Наявність полів level, service,

message, host, trace_id і metadata свідчить про коректне приймання та обробку структури журналу.

Після цього було виконано масове створення журналів подій через HTTP-endpoint /api/v1/logs/batch. У межах цього запиту до системи було передано чотири журнали різних рівнів критичності від різних сервісів. Результат масового створення журналів наведено на рисунку 3.11.


```
Create batch log events
{
    "inserted": 4,
    "ids": [
        "837843f0-9a93-4e82-a3af-ab872827aa6f",
        "619ce21e-44a8-470a-a757-0c6ab7a0911b",
        "ee63eda8-2ab7-44a1-821c-b53fc4f1abe3",
        "6cd62427-7ea1-49c8-a2fa-970a2e40cd6b"
    ]
}
```


Рисунок 3.11 – Результат масового створення журналів подій

Поле inserted має значення 4, що підтверджує успішне додавання чотирьох журналів подій. Також у відповіді повернуто список ідентифікаторів створених записів. Це дає змогу надалі звертатися до конкретних журналів або перевіряти їхню наявність у загальному списку.

На цьому етапі було підтверджено, що система коректно приймає як один журнал події, так і пакет журналів. Усі записи отримали унікальні ідентифікатори, часові мітки та були підготовлені для подальшого зберігання й перегляду.

3.3 Тестування перегляду, фільтрації та пошуку журналів подій

Після створення журналів подій було виконано перевірку їх перегляду та фільтрації. Для цього використано HTTP-endpoint /api/v1/logs, який дозволяє отримувати список журналів із Redis, а також застосовувати фільтри за сервісом і рівнем критичності.

Спочатку було виконано запит на отримання всіх створених журналів. Оскільки перед цим було створено один окремий журнал і чотири журнали в пакетному режимі, загальна кількість записів повинна становити п’ять. Результат перегляду журналів наведено на рисунку 3.12.


```
List logs from Redis
{
    "total": 5,
    "items": [
        {
            "id": "6cd62427-7ea1-49c8-a2fa-970a2e40cd6b",
            "timestamp": "2026-06-12T23:44:24.781495Z",
            "level": "ERROR",
            "service": "billing-service",
            "message": "Payment gateway timeout",
            "host": "worker-02",
            "trace_id": null,
            "metadata": {}
        },
        {
            "id": "ee63eda8-2ab7-44a1-821c-b53fc4f1abe3",
            "timestamp": "2026-06-12T23:44:24.781212Z",
            "level": "CRITICAL",
            "service": "payment-service",
            "message": "Payment provider unavailable",
            "host": "api-02",
            "trace_id": null,
            "metadata": {}
        },
```


Рисунок 3.12 – Результат перегляду списку журналів подій із Redis

З рисунка 3.12 видно, що система повернула п’ять журналів подій. Записи відсортовані у зворотному хронологічному порядку, тобто найновіші журнали відображаються першими. Це підтверджує коректну роботу механізму перегляду журналів із Redis.

Далі було перевірено фільтрацію журналів за назвою сервісу. Для цього було виконано запит із параметром service=auth-service. Результат фільтрації журналів за сервісом наведено на рисунку 3.13.

```
Filter logs by service from Redis
{
    "total": 2,
    "items": [
        {
            "id": "619ce21e-44a8-470a-a757-0c6ab7a0911b",
            "timestamp": "2026-06-12T23:44:24.780750Z",
            "level": "WARNING",
            "service": "auth-service",
            "message": "Too many login attempts",
            "host": "api-01",
            "trace_id": null,
            "metadata": {}
        },
        {
            "id": "03435a52-4864-4020-b7f1-c95b43de3800",
            "timestamp": "2026-06-12T23:44:24.747154Z",
            "level": "ERROR",
            "service": "auth-service",
            "message": "Invalid password for user admin",
            "host": "api-01",
            "trace_id": "trace-demo-001",
            "metadata": {
                "user_id": 101,
                "ip": "192.168.1.10"
            }
        }
    ]
}
```


Рисунок 3.13 – Результат фільтрації журналів за сервісом

У відповіді повернуто два журнали, і в обох поле service має значення auth-service. Це підтверджує, що фільтрація за сервісом працює коректно та дозволяє вибирати записи лише від потрібного джерела.

Також було перевірено фільтрацію журналів за рівнем критичності. Для цього було виконано запит із параметром level=ERROR. Результат фільтрації журналів за рівнем критичності наведено на рисунку 3.14.

З рисунка 3.14 видно, що у відповіді повернуто лише журнали з рівнем ERROR. Це підтверджує правильну роботу фільтрації за рівнем критичності та можливість швидко знаходити помилки серед усіх збережених подій.

```
Filter logs by level from Redis
{
    "total": 2,
    "items": [
        {
            "id": "6cd62427-7ea1-49c8-a2fa-970a2e40cd6b",
            "timestamp": "2026-06-12T23:44:24.781495Z",
            "level": "ERROR",
            "service": "billing-service",
            "message": "Payment gateway timeout",
            "host": "worker-02",
            "trace_id": null,
            "metadata": {}
        },
        {
            "id": "03435a52-4864-4020-b7f1-c95b43de3800",
            "timestamp": "2026-06-12T23:44:24.747154Z",
            "level": "ERROR",
            "service": "auth-service",
            "message": "Invalid password for user admin",
            "host": "api-01",
            "trace_id": "trace-demo-001",
            "metadata": {
                "user_id": 101,
                "ip": "192.168.1.10"
            }
        }
    ]
}
```


Рисунок 3.14 – Результат фільтрації журналів за рівнем критичності

Під час тестування перегляду та фільтрації журналів було підтверджено, що система коректно повертає список збережених подій, підтримує фільтрацію за сервісом та фільтрацію за рівнем.

3.4 Перевірка зберігання журналів у Redis та файловому архіві

Наступним етапом було перевірено, що журнали подій зберігаються не лише в Redis, а й у додатковому файловому архіві storage/logs.jsonl. Такий підхід використовується для демонстрації постійного зберігання журналів і можливості перегляду записів незалежно від оперативного сховища Redis.

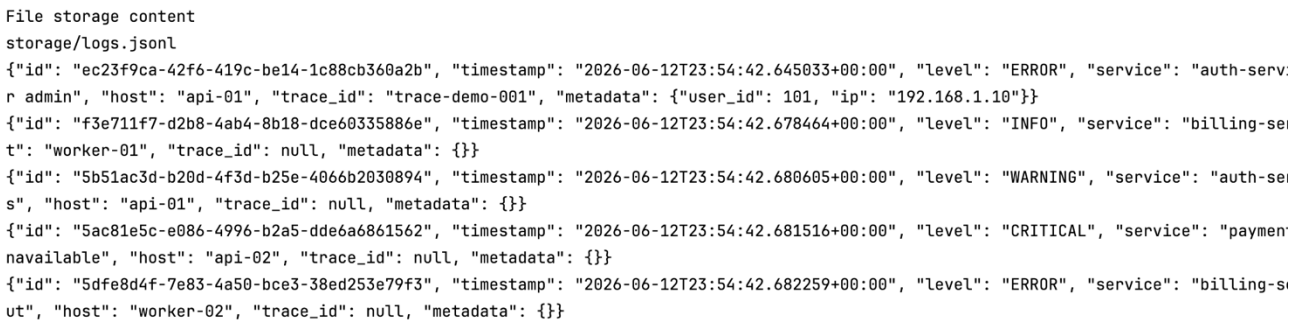
Спочатку було перевірено отримання журналів із файлового архіву за періодом часу. Для цього демонстраційний сценарій виконав запит до HTTP-endpoint /api/v1/logs/archive. Результат перегляду журналів із файлового архіву наведено на рисунку 3.15.

```
Logs from file archive by period
{
    "total": 5,
    "items": [
        {
            "id": "6cd62427-7ea1-49c8-a2fa-970a2e40cd6b",
            "timestamp": "2026-06-12T23:44:24.781495Z",
            "level": "ERROR",
            "service": "billing-service",
            "message": "Payment gateway timeout",
            "host": "worker-02",
            "trace_id": null,
            "metadata": {}
        },
        {
            "id": "ee63eda8-2ab7-44a1-821c-b53fc4f1abe3",
            "timestamp": "2026-06-12T23:44:24.781212Z",
            "level": "CRITICAL",
            "service": "payment-service",
            "message": "Payment provider unavailable",
            "host": "api-02",
            "trace_id": null,
            "metadata": {}
        },
        {
            "id": "619ce21e-44a8-470a-a757-0c6ab7a0911b",
            "timestamp": "2026-06-12T23:44:24.780750Z",
            "level": "WARNING",
```


Рисунок 3.15 – Результат отримання журналів із файлового архіву

У відповіді повернуто п’ять журналів, тобто така сама кількість записів, як і під час перегляду журналів із Redis. Це підтверджує, що створені події були додатково записані у файловий архів і можуть бути отримані через окремий API-запит.

Для додаткової перевірки було переглянуто фактичний вміст файлу storage/logs.jsonl. Результат перегляду останніх рядків файлового архіву наведено на рисунку 3.16.



File storage content
storage/logs.jsonl
{"id": "ec23f9ca-42f6-419c-be14-1c88cb360a2b", "timestamp": "2026-06-12T23:54:42.645033+00:00", "level": "ERROR", "service": "auth-serv
r admin", "host": "api-01", "trace_id": "trace-demo-001", "metadata": {"user_id": 101, "ip": "192.168.1.10"}}
{"id": "f3e711f7-d2b8-4ab4-8b18-dce60335886e", "timestamp": "2026-06-12T23:54:42.678464+00:00", "level": "INFO", "service": "billing-se
t": "worker-01", "trace_id": null, "metadata": {}}
{"id": "5b51ac3d-b20d-4f3d-b25e-4066b2030894", "timestamp": "2026-06-12T23:54:42.680605+00:00", "level": "WARNING", "service": "auth-se
s", "host": "api-01", "trace_id": null, "metadata": {}}
{"id": "5ac81e5c-e086-4996-b2a5-dde6a6861562", "timestamp": "2026-06-12T23:54:42.681516+00:00", "level": "CRITICAL", "service": "paymen
navailable", "host": "api-02", "trace_id": null, "metadata": {}}
{"id": "5dfe8d4f-7e83-4a50-bce3-38ed253e79f3", "timestamp": "2026-06-12T23:54:42.682259+00:00", "level": "ERROR", "service": "billing-s
ut", "host": "worker-02", "trace_id": null, "metadata": {}}

Рисунок 3.16 – Вміст файлового архіву журналів подій

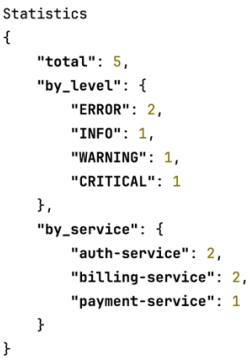
З рисунка 3.16 видно, що кожен журнал зберігається окремим JSON-рядком. У файлі присутні всі створені під час тестування записи, що підтверджує коректну роботу файлового архіву.

Таким чином, було підтверджено, що система зберігає журнали у двох варіантах: в Redis для швидкого доступу та у файлі storage/logs.jsonl для додаткового архівного зберігання.

3.5 Тестування збору статистики та отримання останніх подій

Останнім етапом тестування було перевірено механізми збору статистики та отримання останніх подій. Ці функції дозволяють швидко оцінити загальний стан системи без ручного перегляду всіх журналів.

Для перевірки статистики було виконано запит до HTTP-endpoint /api/v1/stats. Результат отримання статистичних даних наведено на рисунку 3.17.



Statistics
{
    "total": 5,
    "by_level": {
        "ERROR": 2,
        "INFO": 1,
        "WARNING": 1,
        "CRITICAL": 1
    },
    "by_service": {
        "auth-service": 2,
        "billing-service": 2,
        "payment-service": 1
    }
}

Рисунок 3.17 – Результат отримання статистики журналів подій

Поле total має значення 5, що відповідає загальній кількості створених журналів. У блоці by_level показано, що в системі збережено два журнали рівня ERROR, один журнал рівня INFO, один журнал рівня WARNING та один журнал рівня CRITICAL. У блоці by_service показано, що від сервісів auth-service і billing-service отримано по два журнали, а від payment-service - один журнал.

Після цього було перевірено отримання останніх подій з Redis Stream. Для цього виконано запит до HTTP-endpoint /api/v1/stream/latest. Результат отримання останніх подій наведено на рисунку 3.18.


```
Recent stream events
{
    "total": 5,
    "items": [
        {
            "id": "5dfe8d4f-7e83-4a50-bce3-38ed253e79f3",
            "timestamp": "2026-06-12T23:54:42.682259Z",
            "level": "ERROR",
            "service": "billing-service",
            "message": "Payment gateway timeout",
            "host": "worker-02",
            "trace_id": null,
            "metadata": {}
        },
        {
            "id": "5ac81e5c-e086-4996-b2a5-dde6a6861562",
            "timestamp": "2026-06-12T23:54:42.681516Z",
            "level": "CRITICAL",
```


Рисунок 3.18 – Результат отримання останніх подій

З рисунка 3.18 видно, що система повернула п’ять останніх подій. Порядок записів відповідає зворотному хронологічному сортуванню: найновіший журнал відображається першим. Це підтверджує коректну роботу механізму отримання останніх подій із Redis Stream.

Під час тестування було підтверджено, що система коректно формує статистику журналів і дозволяє переглядати останні події. Отримані результати збігаються з кількістю створених журналів, їхніми рівнями критичності та

назвами сервісів, що свідчить про правильну роботу реалізованих механізмів аналізу подій.

3.6 Висновок до третього розділу

У третьому розділі було проведено тестування роботи серверної системи централізованого збору журналів подій. Було перевірено підготовку тестового середовища, запуск FastAPI-застосунку, роботу Redis-сховища та доступність основних API-endpoint. Результат перевірки HTTP-endpoint /api/v1/health підтвердив, що серверна частина працює коректно, а підключення до Redis виконується успішно.

Під час тестування було перевірено приймання журналів подій через REST API. Система коректно обробляла як один журнал, так і пакет журналів, формувала для кожного запису унікальний ідентифікатор, часову мітку та зберігала основні дані події: рівень критичності, назву сервісу, повідомлення, хост, trace_id і додаткові метадані.

Також було протестовано перегляд, фільтрацію та пошук журналів подій. Було підтверджено, що система повертає збережені записи у правильному порядку, а також дозволяє фільтрувати журнали за назвою сервісу та рівнем критичності. Це забезпечує зручний пошук потрібних подій серед загального списку журналів.

Окремо було перевірено зберігання журналів у Redis та додатковому файловому архіві storage/logs.jsonl. Отримані результати показали, що всі створені журнали доступні як через Redis, так і через файловий архів, що підвищує надійність зберігання даних.

Завершальним етапом було перевірено формування статистики та отримання останніх подій. Система коректно підраховувала загальну кількість журналів, групувала їх за рівнями критичності та сервісами, а також повертала останні події з Redis Stream.

Отже, проведене тестування підтвердило працездатність розробленої серверної системи. Система коректно приймає, зберігає, фільтрує, архівує та аналізує журнали подій, що підтверджує правильність реалізації основних функцій централізованого збору логів.

РОЗДІЛ 4. БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ, ОСНОВИ ОХОРОНИ ПРАЦІ

4.1 Працездатність людини – оператора

Під працездатністю оператора розуміють його здатність виконувати професійні завдання з необхідною якістю у визначений термін. Цей показник є комплексною характеристикою, що відображає можливість людини ефективно виконувати свої функції протягом робочого часу. Працездатність визначає не лише продуктивність праці, а й рівень безпеки виконання завдань, особливо у випадках, коли оператор відповідає за складні або відповідальні процеси. Вона формується під впливом багатьох факторів, які умовно поділяють на зовнішні та внутрішні.

До зовнішніх чинників, що впливають на працездатність оператора, належать обсяг і форма інформації, яку він отримує під час роботи, ергономічність робочого місця, характер взаємодії у колективі, а також умови навколишнього середовища. Наприклад, надмірна кількість інформації або її складна структура можуть призводити до перевантаження, що негативно впливає на якість виконання завдань. Невідповідність робочого місця ергономічним вимогам може спричинити швидке настання втоми, а несприятливий мікроклімат або шум – зниження концентрації уваги. Важливу роль відіграє і психологічний клімат у колективі: підтримка з боку колег та керівництва сприяє підвищенню працездатності, тоді як конфлікти або напружені стосунки можуть її знижувати [32].

Внутрішні чинники включають рівень професійної підготовки оператора, його фізичну тренованість, стан здоров’я та емоційну стійкість. Високий рівень підготовки дозволяє швидко орієнтуватися у складних ситуаціях, приймати оптимальні рішення та ефективно використовувати наявні ресурси. Фізична тренованість забезпечує витривалість, а емоційна стійкість допомагає зберігати працездатність навіть у стресових умовах. Особисті якості, такі як

відповідальність, уважність, здатність до самоконтролю, також мають значний вплив на ефективність роботи оператора.

У процесі трудової діяльності оператор проходить різні функціональні стани, які визначають рівень його працездатності. Ці стани змінюються протягом робочого циклу і залежать від інтенсивності навантаження, тривалості роботи, а також від індивідуальних особливостей організму. Відстеження змін функціонального стану дозволяє своєчасно виявляти ознаки втоми та вживати заходів для її попередження.

Виділяють чотири основні фази працездатності: адаптація до роботи, стійка працездатність, субкомпенсація та втома, детальніше на рисунку 4.1. Тривалість кожної з цих фаз залежить від підготовленості оператора, характеру виконуваних завдань, а також від умов праці. Розуміння особливостей кожної фази дозволяє організувати робочий процес таким чином, щоб максимально використовувати періоди високої працездатності та мінімізувати негативні наслідки втоми.

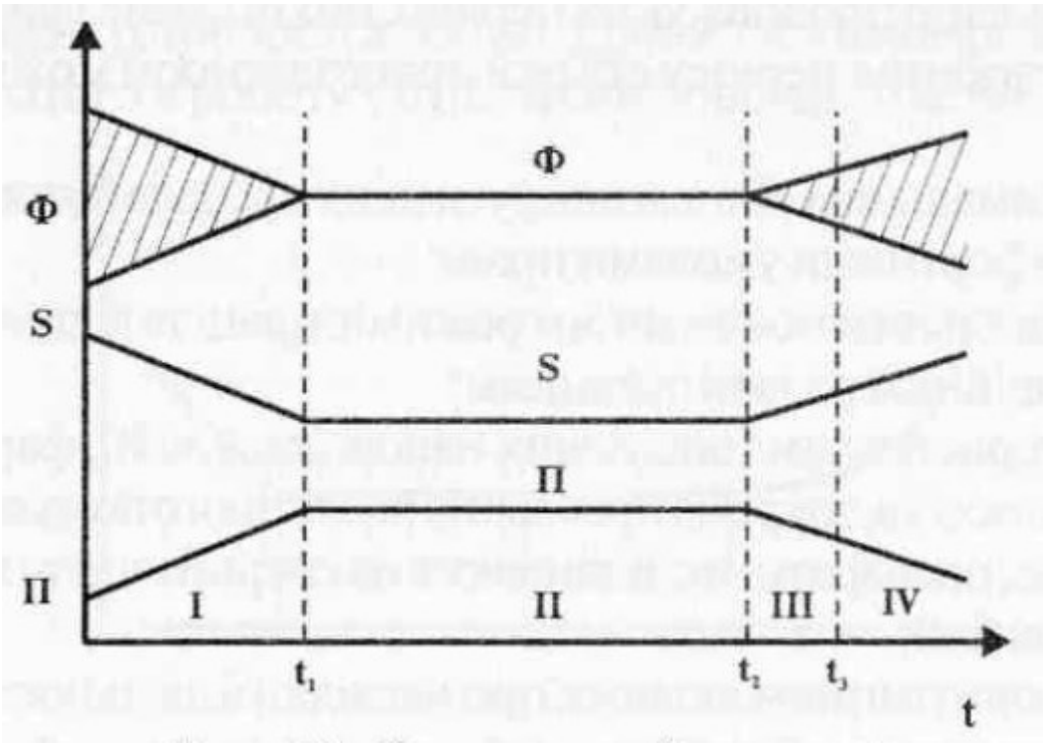


Рисунок 4.1 – Фази працездатності людини

Ф – показник функціонального стану;

Б – помилки роботи;

П – продуктивність праці.

Фаза адаптації ($0–t_1$) – це період, коли організм пристосовується до умов праці. У цей час відбувається налаштування фізіологічних систем на виконання конкретних завдань, формується оптимальний рівень уваги та концентрації. Тривалість цієї фази визначається інтенсивністю роботи, складністю завдань і рівнем підготовки оператора. Скоротити період адаптації дозволяє попереднє тренування, виконання фізичних вправ, адаптація органів чуття до умов праці, а також короткочасне вирішення складних завдань перед початком зміни. Це сприяє швидшому досягненню оптимального функціонального стану.

Фаза стійкої працездатності ($t_1–t_2$) характеризується максимальною якістю виконання завдань при оптимальному функціонуванні організму. У цей період оператор демонструє найвищу продуктивність, мінімальну кількість помилок та стабільний рівень уваги. Тривалість цієї фази залежить від інтенсивності роботи: при динамічній діяльності вона значно довша, ніж при статичній. Емоційний стан оператора суттєво впливає на тривалість стійкої працездатності: позитивні емоції, такі як впевненість у власних силах, задоволення від роботи, подовжують цей період, тоді як негативні – скорочують його.

Підтримати стійку працездатність дозволяють оптимальний рівень психофізіологічної напруги, комфортні умови праці, раціональне чергування праці та відпочинку, емоційне розвантаження, а також інформування про результати роботи. Використання легких стимуляторів (кава, чай) може тимчасово підвищити працездатність, однак надмірне застосування фармакологічних засобів або транквілізаторів призводить до зниження психічної активності та уповільнення реакцій. Важливо забезпечити належний контроль за станом оператора, щоб своєчасно виявляти ознаки перевтоми.

Фаза субкомпенсації ($t_2–t_3$) є початком розвитку втоми. У цей період якість роботи ще залишається на високому рівні, але досягається за рахунок

підвищеного напруження функцій організму. Оператор може відчувати зниження уваги, збільшення кількості помилок, зменшення швидкості реакцій. Якщо не вжити заходів для відновлення працездатності, субкомпенсація переходить у фазу втоми.

Фаза втоми характеризується помітним зниженням якості виконання завдань та погіршенням функціонального стану оператора. Ознаками втоми є зміни частоти пульсу, дихання, а також зниження чутливості органів зору і слуху. Втома може призвести до серйозних помилок, зниження безпеки праці та навіть до виникнення аварійних ситуацій. Тому важливо своєчасно організовувати відпочинок та проводити профілактичні заходи для попередження розвитку втоми.

Після завершення робочого циклу необхідна фаза відновлення працездатності (відпочинку), тривалість якої може варіюватися від кількох хвилин до декількох діб залежно від навантаження, індивідуальних особливостей організму та умов праці. Ефективний відпочинок дозволяє повністю відновити функціональний стан оператора, підвищити його готовність до виконання наступних завдань та знизити ризик виникнення професійних захворювань.

В результаті випливає, що працездатність оператора визначається здатністю виконувати завдання якісно і вчасно. На неї впливають зовнішні (умови праці, інформаційне навантаження, мікроклімат) та внутрішні (підготовка, здоров’я, емоційний стан) чинники. У процесі роботи виділяють фази адаптації, стійкої працездатності, субкомпенсації та втоми. Для підтримки високої працездатності важливо організовувати раціональний режим праці та відпочинку, створювати комфортні умови та слідкувати за станом оператора.

4.2 Рекомендації щодо естетичного оформлення інтер’єру цеху, дільниці

Естетичне оформлення інтер'єру цеху чи дільниці є важливим аспектом, який впливає на продуктивність працівників, їхнє здоров'я та загальне враження від підприємства. Одним із перших кроків у створенні привабливого інтер'єру є вибір кольорової гами. Використання яскравих і теплих кольорів може стимулювати енергію і мотивацію працівників. Наприклад, жовтий колір сприяє підвищенню настрою та креативності, тоді як зелений заспокоює та сприяє концентрації. Важливо уникати надмірно яскравих кольорів, які можуть викликати втому очей та знижувати продуктивність [34].

Наступним важливим аспектом є освітлення. Природне світло є найбільш сприятливим для робочого середовища, тому варто забезпечити максимальний доступ до нього. Це можна зробити за допомогою великих вікон або світлових люків. Якщо природне світло обмежене, варто використовувати якісне штучне освітлення з нейтральними відтінками, яке не викликатиме напруги очей [34]. Освітлення повинно бути рівномірним, без тіней, щоб забезпечити комфортні умови для роботи.

Організація простору також відіграє важливу роль в естетичному оформленні інтер'єру цеху чи дільниці. Простір повинен бути добре організованим та максимально зручним для працівників. Це включає правильне розташування обладнання, робочих місць і зон відпочинку. Важливо уникати захаращення простору, що може призвести до стресу та зниження продуктивності [34].

Значну увагу слід приділити вибору меблів та обладнання. Вони повинні бути зручними, функціональними та відповідати ергономічним вимогам. Використання якісних матеріалів забезпечить довговічність меблів та обладнання, а також сприятиме створенню позитивного враження про підприємство [35]. Ергономічні стільці, столи і робочі станції допоможуть

знизити ризик професійних захворювань і підвищити продуктивність працівників.

Декоративні елементи також можуть значно вплинути на загальний вигляд інтер'єру. Це можуть бути картини, плакати, зелень або інші елементи, які додають індивідуальності та стилю. Використання зелених рослин створює приємну атмосферу, покращує якість повітря і сприяє зниженню стресу. Варто також використовувати декоративні елементи, які відповідають тематиці підприємства або його продукції [35].

Звукова атмосфера є ще одним важливим аспектом естетичного оформлення. Надмірний шум може викликати стрес та знижувати продуктивність працівників [35]. Тому важливо забезпечити звуковий комфорт за допомогою звукоізоляційних матеріалів та обладнання, яке працює тихо. Використання музики або звуків природи може створити приємну атмосферу і покращити настрій працівників.

Температурний режим і вентиляція також важливі для створення комфортного робочого середовища. Оптимальна температура і свіже повітря сприяють підвищенню продуктивності та зниженню втоми. Варто забезпечити достатню кількість вентиляційних отворів або встановити кондиціонери для підтримання комфортної температури та забезпечення циркуляції повітря.

Оформлення зон відпочинку є важливою складовою естетичного оформлення інтер'єру. Вони повинні бути зручними і привабливими, щоб працівники могли повноцінно відпочити під час перерви. Зручні меблі, зелень і приємна атмосфера сприяють відновленню сил і підвищенню мотивації.

Важливим аспектом є також чистота та порядок у цеху чи дільниці. Регулярне прибирання і підтримка чистоти сприяють створенню приємної атмосфери і знижують ризик нещасних випадків та професійних захворювань. Варто встановити системи управління відходами та забезпечити працівників засобами для підтримки чистоти на робочих місцях.

Нарешті, варто звернути увагу на безпеку та доступність інтер'єру. Всі робочі місця повинні бути безпечними і відповідати стандартам безпеки.

Доступність до всіх зон повинна бути забезпечена для всіх працівників, включаючи тих, хто має обмежені фізичні можливості [35]. Це можна досягти за допомогою спеціальних пандусів, ліфтів та інших засобів.

Естетичне оформлення інтер'єру цеху чи дільниці є важливим елементом, який впливає на продуктивність працівників, їхнє здоров'я та загальний імідж підприємства. Правильний вибір кольорової гами, освітлення, організація простору, вибір меблів та обладнання, декоративні елементи, звукова атмосфера, температурний режим і вентиляція, оформлення зон відпочинку, підтримка чистоти і порядку, а також забезпечення безпеки та доступності допоможуть створити привабливий і комфортний робочий простір.

4.3 Висновок до четвертого розділу

У четвертому розділі було розглянуто питання безпеки життєдіяльності та основ охорони праці, зокрема працездатність людини-оператора та рекомендації щодо естетичного оформлення інтер’єру цеху або виробничої дільниці. Перший підрозділ був присвячений аналізу працездатності оператора як здатності людини якісно та своєчасно виконувати професійні завдання протягом робочого часу. Було визначено, що на працездатність впливають зовнішні та внутрішні чинники. До зовнішніх належать умови праці, інформаційне навантаження, ергономічність робочого місця, мікроклімат, шум і психологічний клімат у колективі. До внутрішніх чинників належать рівень професійної підготовки, фізичний стан, здоров’я, емоційна стійкість, уважність і відповідальність працівника.

Також у розділі було розглянуто основні фази працездатності людини-оператора: адаптацію до роботи, стійку працездатність, субкомпенсацію та втому. Було встановлено, що найкращі результати праці досягаються у фазі стійкої працездатності, коли оператор має високий рівень концентрації, мінімальну кількість помилок і стабільну продуктивність. Водночас фази субкомпенсації та втоми свідчать про поступове зниження функціонального

стану організму, що може призводити до помилок, зниження безпеки праці та виникнення аварійних ситуацій. Тому важливим є раціональне чергування праці й відпочинку, створення комфортних умов роботи та своєчасне виявлення ознак перевтоми.

Другий підрозділ охоплює рекомендації щодо естетичного оформлення інтер'єру цеху або дільниці. Було визначено, що правильно організований робочий простір позитивно впливає на продуктивність працівників, їхній психологічний стан, здоров'я та загальне сприйняття підприємства. Особливу увагу було приділено вибору кольорової гами, організації природного та штучного освітлення, розміщенню обладнання, використанню ергономічних меблів, підтриманню чистоти й порядку, забезпеченню вентиляції, комфортного температурного режиму та звукового середовища. Також було підкреслено значення зон відпочинку, декоративних елементів, зелених рослин, безпечності та доступності робочого простору для всіх працівників.

Таким чином, четвертий розділ підкреслює важливість комплексного підходу до організації безпечних, зручних і продуктивних умов праці. Підтримання працездатності оператора, раціональна організація режиму праці та відпочинку, ергономічне облаштування робочого місця, естетичне оформлення виробничого середовища, належне освітлення, вентиляція, чистота та безпека простору сприяють зниженню втоми, підвищенню ефективності праці та створенню комфортних умов для працівників підприємства.

ВИСНОВКИ

Під час виконання кваліфікаційної роботи було реалізовано та протестовано серверну систему централізованого збору журналів подій на основі FastAPI, Redis та додаткового файлового архіву. У роботі досліджено призначення та роль журналів подій у серверних системах, розглянуто підходи до централізованого збору логів, спроєктовано архітектуру системи, реалізовано основні функціональні модулі та проведено практичне тестування працездатності розробленого рішення.

У першому розділі було досліджено роль журналів подій у сучасних серверних системах. Встановлено, що журнали є важливим джерелом інформації про стан програмного забезпечення, помилки, дії користувачів, мережеві взаємодії, безпекові інциденти та роботу окремих сервісів. Було визначено, що ефективне журналювання повинно передбачати не лише збереження повідомлень, а й використання структурованого формату, часових міток, рівнів критичності, ідентифікаторів джерел, trace_id та додаткових метаданих. Це забезпечує можливість подальшого пошуку, аналізу, моніторингу та розслідування інцидентів.

Також у першому розділі було розглянуто основні підходи до централізованого збору, зберігання та аналізу журналів подій. Було встановлено, що централізоване журналювання дозволяє об’єднати події з різних сервісів в одному місці, спростити їх перегляд, фільтрацію, пошук і подальшу обробку. Окрему увагу було приділено існуючим рішенням для збирання, зберігання та аналізу журналів подій, зокрема Elastic Stack, Grafana Loki, Graylog, Wazuh, OpenSearch і Splunk. У результаті аналізу було зроблено висновок, що для розроблення власної серверної системи необхідно забезпечити приймання структурованих журналів, їх валідацію, надійне зберігання, індексацію, пошук, формування статистики та можливість подальшого розширення.

У другому розділі було виконано проєктування та реалізацію серверної системи централізованого збору журналів подій. Розроблена система побудована на основі FastAPI-застосунку, сервісного шару LogService, Redis-сховища та додаткового файлового архіву у форматі JSON Lines. Джерелами журналів подій виступають окремі сервіси, зокрема auth-service, billing-service та notification-service, які передають події до центрального сервера за допомогою HTTP-запитів. Такий підхід дозволяє централізовано збирати журнали з декількох сервісів і забезпечує зручну основу для подальшого аналізу роботи програмної системи.

У системі було реалізовано REST API для створення одного журналу подій, масового додавання записів, перегляду журналів із фільтрацією, отримання окремого запису за ідентифікатором, перегляду останніх подій, формування статистики та пошуку у файловому архіві. Для перевірки структури вхідних даних використовуються Pydantic-схеми, що дозволяє контролювати коректність журналів ще на етапі приймання HTTP-запиту. Кожен запис містить основні атрибути події, зокрема рівень критичності, назву сервісу, повідомлення, хост, trace_id, часову мітку та додаткові метадані.

Основним оперативним сховищем системи було обрано Redis. У ньому реалізовано збереження повних записів журналів у Redis Hash, індексацію за часом у Redis Sorted Set, зберігання останніх подій у Redis Stream та ведення статистичних лічильників. Завдяки цьому система може швидко отримувати журнали за ідентифікатором, повертати список подій у правильному порядку, відображати останні журнали та формувати узагальнену статистику за рівнями критичності й сервісами. Додатково кожен прийнятий журнал дублюється у файловий архів storage/logs.jsonl, що підвищує надійність зберігання даних і дозволяє виконувати пошук незалежно від Redis.

У третьому розділі було проведено практичне тестування розробленої серверної системи. Було перевірено підготовку тестового середовища, запуск FastAPI-застосунку, роботу Redis-сховища та доступність основних API-endpoint. Результат перевірки HTTP-endpoint /api/v1/health підтвердив, що

серверна частина працює коректно, а підключення до Redis виконується успішно. Це засвідчило готовність системи до приймання та обробки журналів подій.

Під час тестування було перевірено створення одного журналу подій і масове додавання декількох записів. Система коректно приймала вхідні дані через REST API, виконувала їх валідацію, формувала унікальні ідентифікатори, додавала часові мітки та зберігала події у Redis і файловому архіві. Також було протестовано перегляд журналів, фільтрацію за назвою сервісу та рівнем критичності, отримання останніх подій і пошук записів у файловому архіві. Отримані результати підтвердили, що система правильно повертає збережені журнали та забезпечує зручний доступ до потрібних подій.

Окремо було перевірено формування статистики журналювання. Система коректно підраховувала загальну кількість журналів, групувала події за рівнями критичності та сервісами, а також повертала останні записи з Redis Stream. Це підтвердило правильність реалізації механізмів статистичної обробки та оперативного перегляду нових подій. Наявність одночасного зберігання у Redis і файловому архіві забезпечує поєднання швидкого доступу до актуальних даних із додатковим постійним збереженням журналів.

У результаті виконання кваліфікаційної роботи було успішно реалізовано та перевірено на практиці серверну систему централізованого збору журналів подій. Розроблене рішення забезпечує приймання, валідацію, збереження, індексацію, фільтрацію, пошук, архівування та статистичний аналіз журналів. Використання FastAPI дозволило створити зручний REST API для взаємодії із системою, Redis забезпечив швидке оперативне зберігання та доступ до подій, а файловий архів у форматі JSON Lines підвищив надійність збереження даних. Проведене тестування підтвердило працездатність системи та правильність реалізації її основних функцій. Запропоноване рішення може бути використане як основа для подальшого розвитку систем моніторингу, централізованого аналізу журналів подій і контролю стану серверної інфраструктури.

ПЕРЕЛІК ДЖЕРЕЛ

1. K Kent, K., & Souppaya, M. (2006). Guide to computer security log management (NIST Special Publication 800-92). National Institute of Standards and Technology. https://csrc.nist.gov/pubs/sp/800/92/final

2. Kharchenko, A., Bodnarchuk, I., & Yatcyshyn, V. (2014). The method for comparative evaluation of software architecture with accounting of trade-offs. American Journal of Information Systems, 2(1), 20-25.

3. Bodnarchuk, I., Lisovyi, V., Kharchenko, O., & Galai, I. (2018, September). Adaptive method for assessment and selection of software architecture in flexible techniques of design. In 2018 IEEE 13th International Scientific and Technical Conference on Computer Sciences and Information Technologies (CSIT) (Vol. 1, pp. 292-297). IEEE.

4. Gerhards, R. (2009). The Syslog Protocol (RFC 5424). Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/rfc5424

5. Okmianski, A. (2009). Transmission of Syslog Messages over UDP (RFC 5426). Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/rfc5426

6. Miao, F., Ma, Y., & Salowey, J. (2009). Transport Layer Security (TLS) Transport Mapping for Syslog (RFC 5425). Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/rfc5425

7. Ihor, B., Oleksii, D., Aleksandr, K., Nataliia, K., Oleksandr, M., & Volodymyr, P. (2019, January). Multicriteria choice of software architecture using dynamic correction of quality attributes. In International Conference on Computer Science, Engineering and Education Applications (pp. 419-427). Cham: Springer International Publishing.

8. OpenTelemetry. (n.d.). OpenTelemetry Logs Data Model. https://opentelemetry.io/docs/specs/otel/logs/data-model/

9. OpenTelemetry. (n.d.). OpenTelemetry Protocol Specification. https://opentelemetry.io/docs/specs/otlp/

10. Kharchenko, A., Raichev, I., Bodnarchuk, I., & Matsiuk, O. (2021, October). The Survey of Global Software Design Processes. In 2021 IEEE 8th International Conference on Problems of Infocommunications, Science and Technology (PIC S&T) (pp. 291-294). IEEE.

11. Elastic. (n.d.). Elastic Common Schema documentation. https://www.elastic.co/docs/reference/ecs

12. Волович, В., Береженко, Б. М., & Боднарчук, І. О. (2022). Задача проєктування програмної архітектури в процесах забезпечення якості. Матеріали X науково-технічної конференції „Інформаційні моделі, системи та технології “Тернопільського національного технічного університету імені Івана Пулюя, 104-106.

13. Боднарчук, І., Харченко, О., Хоміцький, Б., & Шимчук, Г. (2019). Проектування архітектури програмних систем в проектах з гнучкими методами управління. Матеріали XXI наукової конференції Тернопільського національного технічного університету імені Івана Пулюя, 46-48.

14. Elastic. (n.d.). Data streams documentation. https://www.elastic.co/docs/manage-data/data-store/data-streams

15. Grafana Labs. (n.d.). Grafana Loki documentation. https://grafana.com/docs/loki/latest/

16. Kharchenko, A., Halay, I., Zagorodna, N., & Bodnarchuk, I. (2015, September). Trade-off optimal decision of the problem of software system architecture choice. In 2015 Xth International Scientific and Technical Conference" Computer Sciences and Information Technologies"(CSIT) (pp. 198-205). IEEE.

17. Orobchuk, B., Buniak, O., Sysak, I., Babiuk, S., Bodnarchuk, I., & Koval, V. (2024). Development of Software for the Implementation of Automated Reserve Input Modes Operation. In CITI (pp. 316-336).

18. Graylog. (n.d.). Graylog documentation. https://go2docs.graylog.org/

19. Wazuh. (n.d.). Wazuh documentation. https://documentation.wazuh.com/current/index.html

20. Fluent Bit. (n.d.). Buffering documentation. https://docs.fluentbit.io/manual/data-pipeline/buffering

21. Apache Kafka. (n.d.). Apache Kafka documentation. https://kafka.apache.org/documentation/

22. OpenSearch. (n.d.). OpenSearch Data Prepper documentation. https://docs.opensearch.org/latest/data-prepper/

23. Splunk. (n.d.). Set a retirement and archiving policy. Splunk Enterprise Documentation. https://help.splunk.com/en/splunk-enterprise/administer/manage-indexers-and-indexer-clusters/10.2/back-up-and-archive-your-indexes/set-a-retirement-and-archiving-policy

24. European Parliament and Council of the European Union. (2016). Regulation (EU) 2016/679 of the European Parliament and of the Council: General Data Protection Regulation. Official Journal of the European Union. https://eur-lex.europa.eu/eli/reg/2016/679/oj/eng

25. HTTP API vs. REST API: Key Differences Explained Simply. (n.d.). testomat.io. https://testomat.io/blog/http-api-vs-rest-api-key-differences-explained/

26. JSON. (n.d.). JSON. https://www.json.org/

27. FastAPI - FastAPI. (n.d.). FastAPI - FastAPI. https://fastapi.tiangolo.com/

28. Redis. (2025, February 10). Redis - Real-time data for agents & apps. https://redis.io/

29. Karpinski, M., Revniuk, O., Tymoshchuk, D., Kozak, R., Tokkuliyeva, A. Fuzzy logic system as a component of the web application security information system. Ceur Workshop Proceedings, 2025, 4042, pp. 308–315

30. Welcome to Pydantic. (n.d.). Pydantic Docs. https://pydantic.dev/docs/validation/latest/get-started/

31. HTTP Endpoints. (n.d.). Mock APIs - Free REST & SOAP APIs for Devs & QA. https://beeceptor.com/docs/concepts/http-endpoints/

32. Бедрій Я.І. Основи охорони праці : навч. посіб. 4-е вид. перероб. і доп. — Тернопіль : Навчальна книга – Богдан, 2018. — 240 с. — Розділ 1.2.3.

33. Желібо Є. П., Сагайдак І. С. Безпека життєдіяльності. Навчальний посібник для аудиторної та практичної роботи. К.:ЕКОМЕН. 2011. 200 с.

34. 7 секретів від дизайнера інтер'єру робочих просторів | Продизайн. (n.d.). Retrieved from https://prodesign.in.ua/2023/08/7-sekretiv-vid-dyzajnera-interyeru-robochyh-prostoriv/

35. 5 підказок щодо оформлення інтер'єру робочого місця - Видавництво ArtHuss. (n.d.). Retrieved from https://www.arthuss.com.ua/books-blog/5-pidkazok-shchodo-oformlennya-interyeru-robochoho-mistsya