Міністерство освіти і науки України
Луцький національний технічний університет
Факультет комп'ютерних та інформаційних технологій
Кафедра інженерії програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «БАКАЛАВР»

РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ АВІАДІСПЕТЧЕРСКОЇ З ВИКОРИСТАННЯМ SPRING, KAFKA, WEBSOCKET ТА NUXT.JS

DEVELOPMENT OF SOFTWARE FOR AIR TRAFFIC CONTROL USING SPRING, KAFKA, WEBSOCKET AND NUXT.JS

спеціальність 121 «Інженерія програмного забезпечення»
освітня програма «Інженерія програмного забезпечення»

Виконав: здобувач вищої освіти
групи ІПЗс-21
Зайцев Олександр Дмитрович

Керівник:
Ліщина Наталія Миколаївна

Кваліфікаційну роботу
допущено до захисту
«10» 06 2025 р.
Гарант освітньої програми:
к.т.н., доцент
Ліщина Наталія Миколаївна

Луцьк – 2025 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет *комп'ютерних та інформаційних технологій*
Кафедра *інженерії програмного забезпечення*
Ступінь вищої освіти *бакалавр*
Галузь знань: *12 «Інформаційні технології»*
Спеціальність: *121 «Інженерія програмного забезпечення»*
Освітня програма: «*Інженерія програмного забезпечення»*

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____
«20» 12 2024 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ВИЩОЇ ОСВІТИ

Зайцеву Олександру Дмитровичу
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи *«Розробка програмного забезпечення для авіадиспетчерскої з використанням Spring, Kafka WebSocket та Nuxt.js»*

Керівник роботи: *к.т.н., доцент Ліщина Н.М.*
затверджені наказом закладу вищої освіти від *«19» грудня 2024 р. № 474/01-02*
2. Строк подання здобувачем вищої освіти кваліфікаційної роботи бакалавра *«10» червня 2025 р.*
3. Вихідні дані до роботи: *Spring, Kafka WebSocket, Nuxt.js, IntelliJ IDEA, методичні вказівки до виконання кваліфікаційної роботи бакалавра.*

4. Зміст розрахунково-пояснювальної записки (перелік питань, що потрібно розробити): *аналіз сучасного стану систем авіадиспетчерського керування, сформулювати завдання створення мікросервісного застосунку з реєстрацією та візуалізацією маршрутів літаків. Сформувано вимоги до обробки повідомлень у реальному часі, обґрунтовано вибір технологій Kafka, Spring Boot, WebSocket, Nuxt.js. Реалізація сервісів Office, Ship та клієнтської частини, обґрунтування підходів до захисту, аналіз масштабованості, тестування та налагодження системи.*

5. Перелік графічного матеріалу: *5 рисунків.*

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз предметної області	*Ліщина Н. М.*		
Специфікація вимог до розробленої системи	*Ліщина Н. М.*		
Розробка об'єкта проектування	*Ліщина Н. М.*		
Нормоконтроль	*Вознюк А. В.*		
Гарант ОП	*Ліщина Н. М.*		
Показник запозичень тексту	3,22 %		
Академічна доброчесність	*Ліщина Н. М.*		

7. Дата видачі завдання *«20» грудня 2024 р.*

№	Назва етапів кваліфікаційної роботи бакалавра	Строк виконання етапів роботи	Примітка
1	Огляд літературних джерел по темі кваліфікаційної роботи бакалавра	до 04.02.2025 р.	Виконано
2	Аналіз проблеми розробки та впровадження об'єкту проектування	до 01.03.2025 р.	Виконано
3	Обґрунтування вибору шляхів, технологій і засобів вирішення поставленого завдання	до 15.03.2025 р.	Виконано
4	Розробка функціонально-структурної схеми роботи об'єкта проектування	до 29.03.2025 р.	Виконано
5	Практична реалізація об'єкта проектування та розробка бази даних	до 26.04.2025 р.	Виконано
6	Тестування та налагодження об'єкта проектування	до 03.05.2025 р.	Виконано
7	Здача чистового варіанту кваліфікаційної роботи бакалавра на кафедру	до 10.06.2025 р.	Виконано

Здобувач вищої освіти _____ Олександр ЗАЙЦЕВ

Керівник кваліфікаційної роботи _____ Наталія ЛІЩИНА

АНОТАЦІЯ

Зайцев О. Д. Розробка програмного забезпечення для авіадиспетчерської з використанням Spring, Kafka,WebSocket та Nuxt.js. Рукопис. Кваліфікаційна робота бакалавра ОП «Інженерія програмного забезпечення» спеціальності «Інженерія програмного забезпечення». Луцький національний технічний університет. Луцьк, 2025.

Кваліфікаційна робота бакалавра складається зі вступу, трьох розділів, висновків, списку використаних джерел.

У першому розділі проведено аналіз предметної області, визначено основні вимоги до системи авіадиспетчерського управління, охарактеризовано типи даних, що підлягають обробці, та обґрунтовано доцільність застосування мікросервісної архітектури.

У другому розділі здійснено проєктування архітектури системи: описано структуру взаємодії сервісів, обрано оптимальні технології (Spring Boot, Apache Kafka, WebSocket, Nuxt.js), сформовано схеми інформаційних потоків та визначено механізми обміну повідомленнями між компонентами.

У третьому розділі детально описано процес реалізації програмного забезпечення: створення мікросервісів, налагодження взаємодії через Kafka і WebSocket, впровадження системи логування, автоматизації збирання та розгортання за допомогою GitHub, Docker і CI/CD, а також представлено підходи до тестування, підтримки та документування проєкту.

Результатом роботи стало створення прототипу програмного комплексу, який може використовуватись у навчальних цілях або як основа для подальшого впровадження в реальні умови.

Ключові слова: авіадиспетчерська система, мікросервісна архітектура, Spring Boot, Kafka, WebSocket, Nuxt.js, інформаційна система, система реального часу.

ABSTRACT

Zaitsev O. Development of Software for Air Traffic Control Using Spring, Kafka, WebSocket and Nuxt.js. Manuscript. Qualification work of the bachelor's program "Software engineering" in the specialty "Software engineering". Lutsk National Technical University. Lutsk, 2025.

The bachelor's qualification work consists of an introduction, three chapters, conclusions, and a list of references.

The first chapter presents an analysis of the subject area, defines the main requirements for the air traffic control system, characterizes the types of data to be processed, and substantiates the feasibility of using a microservice architecture.

The second chapter covers the system architecture design: the structure of service interaction is described, optimal technologies are selected (Spring Boot, Apache Kafka, WebSocket, Nuxt.js), data flow diagrams are developed, and the mechanisms of message exchange between components are defined.

The third chapter provides a detailed description of the software implementation process: the development of microservices, configuration of interaction via Kafka and WebSocket, implementation of a logging system, automation of building and deployment using GitHub, Docker, and CI/CD, as well as approaches to testing, maintenance, and documentation.

The result of the work is the creation of a prototype of a software system that can be used for educational purposes or as a basis for further implementation in real-world conditions.

Keywords: air traffic control system, microservice architecture, Spring Boot, Kafka, WebSocket, Nuxt.js, information system, real-time system.

ЗМІСТ

ВСТУП

Актуальність теми. Авіаційна галузь є однією з найтехнологічніших сфер, де ефективність та безпека залежить від точності, швидкості обробки інформації та надійності комунікації між всіма учасниками повітряного руху. Однією з ключових ланок у цьому процесі є авіадиспетчерська служба, яка забезпечує координацію та контроль за польотами повітряних суден у реальному часі.

Сучасні системи авіадиспетчерського управління потребують високого рівня автоматизації, обробки великих обсягів даних та надійного обміну повідомленнями між різними компонентами системи. Традиційні монолітні рішення поступово втрачають свою ефективність, поступаючись місцем гнучким і масштабованим підходам на основі мікросервісної архітектури.

Аналіз сучасного стану вказує на наявність програмних рішень, здатних обробляти авіаційну інформацію в реальному часі. Проте багато з них є закритими, складними для адаптації або потребують значних фінансових ресурсів. Це створює передумови для розробки нових програмних продуктів, побудованих на відкритих та гнучких технологіях.

Світові тенденції демонструють активне впровадження таких інструментів, як Spring Boot (для створення backend-сервісів), Kafka (як брокер повідомлень), WebSocket (для двостороннього зв’язку в реальному часі) та Nuxt.js (для розробки швидких і адаптивних веб-інтерфейсів).

Актуальність роботи полягає у створенні універсального програмного забезпечення для авіадиспетчерської, яке може застосовуватись як у навчальних цілях (тренажери, симулятори), так і як прототип для подальших промислових рішень.

Взаємозв’язок з іншими роботами полягає у використанні типових архітектурних шаблонів, які застосовуються у галузі інформаційних систем управління та IoT-рішень.

Метою кваліфікаційної роботи є розробка програмного забезпечення для авіадиспетчерської з використанням технологій Spring, Kafka, WebSocket та Nuxt.js, що забезпечує обробку, передачу та візуалізацію інформації про повітряний рух у реальному часі.

Об’єктом кваліфікаційної роботи у даній роботі є програмне забезпечення для авіадиспетчерського управління повітряним рухом**.**

Предметом кваліфікаційної роботи є архітектурні, технічні та інтерфейсні рішення, що забезпечують ефективну обробку, передачу та візуалізацію авіаційної інформації в режимі реального часу.

Завдання до виконання кваліфікаційної роботи:

- дослідити сучасні підходи до архітектури програмних систем у сфері повітряного руху;
- обґрунтувати вибір технологічного стеку для побудови системи;
- спроєктувати архітектуру програмного забезпечення;
- реалізувати backend-сервіси на базі Spring Boot з використанням Kafka та WebSocket;
- розробити клієнтський інтерфейс з використанням Nuxt.js;
- забезпечити реєстрацію телеметрії, обмін повідомленнями між сервісами та візуалізацію даних;
- провести тестування окремих компонентів і системи в цілому;
- здійснити загальний аналіз ефективності розробленого рішення.

РОЗДІЛ 1
АНАЛІЗ АВІАДИСПЕТЧЕРСЬКИХ ІНФОРМАЦІЙНИХ СИСТЕМ І ПОСТАНОВКА ЗАВДАНЬ НА КВАЛІФІКАЦІЙНУ РОБОТУ

1.1 Аналіз сучасного стану проблеми

На сучасному етапі розвитку авіаційної галузі спостерігається постійне зростання інтенсивності повітряного руху. Це обумовлює підвищення вимог до систем управління повітряним простором, зокрема до роботи авіадиспетчерських служб. Оперативність, точність, ефективна координація та своєчасне реагування на зміну ситуацій у повітрі є ключовими умовами безпечного функціонування авіаційної інфраструктури. Саме тому інформаційні системи, які підтримують роботу диспетчерів, повинні відповідати високим стандартам продуктивності та надійності.

На сьогодні в авіадиспетчерській практиці активно використовуються спеціалізовані програмні комплекси, що забезпечують отримання, обробку та візуалізацію даних про переміщення повітряних суден. Більшість із них є комерційними закритими рішеннями, які розробляються великими компаніями, зокрема SkySoft-ATM, Indra, Frequentis та інші. Такі системи орієнтовані на масштабне впровадження в умовах реальних аеронавігаційних підприємств, однак вони мають низку обмежень. Серед них – складність у налаштуванні, висока вартість придбання і супроводу, недостатня гнучкість щодо модифікації функціоналу відповідно до специфічних потреб користувача. До того ж, у навчальному середовищі або при розробці прототипів, подібні продукти часто виявляються недоступними.

В умовах сучасних технологічних тенденцій важливим є використання гнучких підходів до побудови програмного забезпечення. Зокрема, все більшого поширення набуває мікросервісна архітектура, яка забезпечує розділення системи на окремі незалежні компоненти [1]. Вона дозволяє легко масштабувати застосунок, адаптувати його до змін, а також поетапно впроваджувати нові функції без втручання у вже реалізовану логіку.

У зв’язку з цим виникає потреба у створенні програмного продукту, що поєднує переваги сучасних технологій – таких як Spring Boot для реалізації серверної логіки, Apache Kafka для обміну повідомленнями між сервісами у реальному часі, WebSocket для постійного двостороннього зв’язку між клієнтом і сервером, а також Nuxt.js для створення адаптивного, інтерактивного користувацького інтерфейсу. Рішення, побудоване на основі вказаних технологій, дозволить створити гнучку, масштабовану і доступну систему для моделювання роботи авіадиспетчерської служби [2].

Отже, актуальність даного дослідження зумовлена необхідністю розробки відкритого, сучасного, технологічного рішення, яке може бути застосоване як у навчальних цілях, так і як основа для подальшої розробки професійних систем управління повітряним рухом.

1.2 Постановка завдання на кваліфікаційну роботу бакалавра

З урахуванням вищезазначеного, метою цієї кваліфікаційної роботи є розробка інформаційної системи авіадиспетчерської, яка дозволяє моделювати роботу диспетчера у режимі реального часу. Такий застосунок повинен наочно демонструвати обробку потоків телеметричної інформації, її відображення, а також забезпечувати інтерактивну взаємодію користувача з інтерфейсом системи.

Завдання до виконання кваліфікаційної роботи:

- дослідити сучасні підходи до архітектури програмних систем у сфері повітряного руху;
- обґрунтувати вибір технологічного стеку для побудови системи;
- спроєктувати архітектуру програмного забезпечення;
- реалізувати backend-сервіси на базі Spring Boot з використанням Kafka та WebSocket;
- розробити клієнтський інтерфейс з використанням Nuxt.js;

– забезпечити реєстрацію телеметрії, обмін повідомленнями між сервісами та візуалізацію даних;

– провести тестування окремих компонентів і системи в цілому;

– здійснити загальний аналіз ефективності розробленого рішення.

Таким чином, дана робота охоплює повний цикл створення програмного забезпечення: від аналізу проблемної області до реалізації, тестування і опису результатів. Вона має практичне спрямування та може бути використана як у навчальному процесі, так і для подальшого розвитку як частина більших інформаційних систем.

Висновки до розділу 1

У цьому розділі було проаналізовано стан сучасних систем, що використовуються у сфері авіадиспетчеризації. Описано основні тенденції у розробці подібного програмного забезпечення, а також технології, що дозволяють створити сучасне рішення з високими показниками продуктивності. Визначено актуальність роботи та окреслено основну мету дослідження. Сформовано загальну концепцію системи, яка буде розроблятися у подальших розділах кваліфікаційної роботи.

РОЗДІЛ 2
СПЕЦИФІКАЦІЯ ВИМОГ ДО РОЗРОБЛЮВАНОЇ СИСТЕМИ

2.1 Аналіз, визначення вимог до розроблюваного програмного забезпечення та проектування програмного забезпечення

На сучасному етапі розвитку інформаційних технологій усе більшої актуальності набувають програмні архітектури, здатні забезпечити не лише ефективне функціонування у стандартних умовах, але й витримувати навантаження, пов’язані з обробкою подій у реальному часі. Саме тому під час розробки програмного забезпечення для управління повітряним рухом було вирішено застосувати мікросервісну архітектуру, яка найбільш повно відповідає потребам такого класу систем.

Мікросервісна модель побудови програмного забезпечення передбачає поділ загального функціоналу на окремі, відокремлені один від одного компоненти (сервіси), кожен з яких виконує чітко визначене завдання. Такий підхід забезпечує високу ступінь модульності, дозволяє здійснювати незалежне тестування, оновлення та масштабування кожного з сервісів, а також спрощує процес виявлення й усунення помилок.

У рамках розробленої системи було виокремлено кілька базових мікросервісів:

- сервіс аеропорту, що відповідає за прийом даних про переміщення, висоту, швидкість та ідентифікатори повітряних суден, розподіляючи маршрути на вільні борти;
- сервіс літаків, який трансформує, перевіряє та агрегує отриману інформацію від аеропортів, приймаючи маршурти, та починаючи їх виконання;
- клієнтська частина, реалізована засобами фреймворку Nuxt.js, яка забезпечує інтерфейс керування та візуалізацію.

Для забезпечення взаємодії між мікросервісами обрано модель асинхронної комунікації на основі брокера повідомлень Apache Kafka. Такий

механізм дозволяє досягти високої пропускної здатності при передаванні повідомлень між компонентами та забезпечує можливість обробки значного обсягу даних без зниження швидкодії.

З метою кращого розуміння структури розроблюваної системи, а також взаємодії між її окремими компонентами, доцільним є подання загальної архітектури у вигляді UML-діаграми компонентів. На діаграмі наочно зображено мікросервіси, засоби обміну повідомленнями, а також зв’язки між серверною та клієнтською частинами системи. Вона демонструє напрямки передачі даних, використання брокера Kafka як центрального елемента подієвої взаємодії, а також роль WebSocket у забезпеченні зв’язку з інтерфейсом диспетчера. Умовне графічне представлення наведено на рисунку 2.1.

Рисунок 2.1 – UML-діаграма компонентної архітектури системи диспетчеризації повітряного руху

Загальна архітектурна модель системи передбачає поділ на два логічних шари: серверну частину (бекенд) та клієнтську частину (фронтенд). Такий підхід забезпечує незалежність між інтерфейсом та логікою обробки даних, що спрощує процес внесення змін і дозволяє гнучко адаптувати систему до нових вимог.

Основні компоненти:

- серверна частина (бекенд) – реалізує логіку обробки телеметрії, виявлення конфліктів, генерації команд та забезпечення взаємодії між мікросервісами;
- Kafka-брокер – виступає посередником для обміну подіями між мікросервісами, дозволяючи відокремити логіку обробки від логіки передачі даних;
- клієнтська частина (фронтенд) – відповідає за візуалізацію інформації, зокрема відображення позицій літаків, повідомлень, повідомлень про конфлікти та інших елементів інтерфейсу;
- сервіс WebSocket – забезпечує миттєве надсилання оновлень про зміну ситуації у повітряному просторі до клієнта.

Обмін повідомленнями між сервісами організовано за принципом подієвої взаємодії, що дозволяє досягти високої відмовостійкості та гнучкості. У разі виходу з ладу одного з компонентів, система зберігає здатність функціонувати, не блокуючи інші процеси.

На етапі планування було визначено як функціональні, так і нефункціональні вимоги до розроблюваної системи. Вони слугували основою для прийняття рішень під час вибору технологій, побудови архітектури та структури взаємодії компонентів.

Функціональні вимоги:

- обробка телеметричних повідомлень у режимі реального часу;
- візуалізація повітряних суден на карті з оновленням положення;
- підтримка управління літальними апаратами з боку диспетчера;
- фіксація важливих подій та логування дій користувача.

Нефункціональні вимоги:

- масштабованість – можливість розгортання кожного мікросервісу на окремому сервері або контейнері;
- висока доступність – система повинна зберігати працездатність у випадку часткових збоїв;
- продуктивність – забезпечення мінімальних затримок при передачі повідомлень;
- інтуїтивність інтерфейсу – користувач (диспетчер) має змогу швидко орієнтуватися у функціоналі без тривалого навчання.

Особливу увагу було приділено ергономіці інтерфейсу: елементи управління повинні бути логічно згруповані, повідомлення – інформативними, а кольорова палітра – не перевантаженою.

2.2 Обґрунтування вибору технологій

Під час розробки програмного забезпечення будь-якого рівня складності одним із найважливіших етапів є визначення доцільного та обґрунтованого технологічного стеку, який буде використовуватися на всіх етапах життєвого циклу продукту – від початкової ідеї до розгортання і подальшої підтримки. З огляду на специфіку кваліфікаційної роботи, яка полягала в реалізації системи, здатної в режимі реального часу обробляти телеметричні дані з подальшою їх обробкою, маршрутизацією та візуалізацією, до вибору технологій були висунуті суворі вимоги.

Першочергово йшлося про необхідність забезпечення стабільної, передбачуваної поведінки програмного забезпечення в умовах великого навантаження, а також низької затримки передачі даних. Система, що імітує роботу авіадиспетчерської, має обробляти велику кількість подій у реальному часі, при цьому зберігаючи послідовність повідомлень, гарантуючи доставку та підтримуючи логіку обробки потоків. Отже, вибрані технології мали бути

не лише сучасними й актуальними, але й перевіреними в контексті подібних навантажень.

Однією з ключових платформ, що стала основою серверної частини, є Spring Boot – високорівневий Java-фреймворк, який дозволяє створювати продуктивні, масштабовані вебзастосунки з мінімальним обсягом конфігураційного коду. Він надає готові до використання модулі для роботи з HTTP-запитами, REST-контролерами, WebSocket-з'єднанням, потоковою обробкою даних, плануванням завдань, взаємодією з брокерами повідомлень, підключенням до баз даних тощо. Найважливішою перевагою Spring Boot у контексті даного проєкту стала його висока гнучкість у поєднанні з модульністю, що дозволило створити кожен сервіс як незалежний, ізольований блок, з можливістю масштабування та налаштування без ризику вплинути на інші частини системи [3].

Ще одним вагомим аргументом на користь Spring Boot стала його активна підтримка інверсії керування (IoC) та ін'єкції залежностей (DI) [4]. Завдяки цьому було забезпечено чисту архітектуру застосунку, що відповідає принципам SOLID, а також стало можливим легке тестування та рефакторинг окремих компонентів [5]. Важливим є також те, що Spring Boot підтримує анотаційно-орієнтований підхід, що значно спрощує конфігурацію та дозволяє розробнику зосередитися безпосередньо на бізнес-логіці.

Проте жоден мікросервіс не може функціонувати ізольовано без надійного транспортного шару для взаємодії з іншими компонентами системи. У цьому проєкті таку роль виконує Apache Kafka – одна з найпотужніших сучасних платформ для потокової обробки повідомлень. Kafka є брокером повідомлень, який реалізує асинхронну модель обміну даними типу «publisher–subscriber», що дозволяє сервісам передавати інформацію одне одному без прямої залежності [6]. Такий підхід істотно знижує зв'язність між компонентами системи, а отже, підвищує гнучкість, стійкість до збоїв і можливість незалежного масштабування окремих мікросервісів. Наприклад, сервіс генерації телеметрії може надсилати тисячі повідомлень щосекунди, не

чекаючи, поки їх буде оброблено – Kafka гарантує доставку, зберігання і ретрансляцію цих повідомлень.

Окремої уваги заслуговує реалізація високої пропускної здатності Kafka: система здатна витримувати великі обсяги даних навіть у складних розподілених середовищах. Завдяки тому, що Kafka працює з топіками, у які публікуються повідомлення, і дозволяє підписникам споживати дані незалежно один від одного, було досягнуто повної масштабованості як у горизонтальній площині (додавання нових консьюмерів), так і у вертикальній (підвищення продуктивності кожного сервісу). У даному проєкті Kafka також виступає логічним центром усієї архітектури, оскільки всі критично важливі повідомлення проходять саме через цей транспортний рівень.

Щоб забезпечити миттєву передачу повідомлень на фронтенд, було використано протокол WebSocket, який дає змогу підтримувати постійне з’єднання між сервером і клієнтом [7]. Це дає змогу, зокрема, уникнути витрат ресурсів на регулярні HTTP-запити (polling) з боку клієнта, забезпечуючи натомість подієво-орієнтовану модель обміну даними, де повідомлення надходять лише тоді, коли з’являються нові події. У системі моніторингу авіаційного простору це критично важливо, оскільки дозволяє без затримок оновлювати інформацію про рух літаків, зміну координат, виявлення конфліктних ситуацій тощо.

Щодо інтерфейсної частини, то було вирішено використовувати Nuxt.js – високопродуктивний фреймворк на основі Vue.js, який поєднує зручність створення SPA (single-page application) з можливостями server-side rendering [8]. Завдяки цьому забезпечується швидке завантаження сторінок, покращене SEO та зниження навантаження на клієнтські пристрої, що особливо важливо для роботи диспетчерів, які можуть використовувати як сучасні комп’ютери, так і слабші ноутбуки або планшети. Крім того, Nuxt.js має вбудовані механізми маршрутизації, розподілу сторінок, компонування інтерфейсу та управління станом, що дозволило реалізувати інтуїтивно

зрозумілий, адаптивний та привабливий інтерфейс користувача, орієнтований на практичну ефективність у польових умовах.

Синергія згаданих технологій – Spring Boot, Apache Kafka, WebSocket, Nuxt.js – дозволила побудувати сучасну, модульну, реактивну мікросервісну архітектуру, здатну до гнучкого масштабування, ефективного управління подіями та зручної взаємодії з кінцевим користувачем. При цьому всі компоненти мають активну спільноту, документацію, стабільні релізи та доведену ефективність у комерційних і промислових проєктах. Таким чином, вибір технологій є повністю обґрунтованим як з інженерної, так і з академічної точки зору.

Висновки до розділу 2

У ході проєктування системи управління повітряним рухом було здійснено комплексний аналіз можливих архітектурних рішень, а також обґрунтовано вибір мікросервісного підходу як найбільш ефективного в контексті поставлених завдань. Обраний стек технологій дозволяє реалізувати систему, здатну обробляти велику кількість подій у реальному часі, забезпечуючи стабільну роботу та зручність для кінцевого користувача.

РОЗДІЛ 3
РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ АВІАДИСПЕТЧЕРСЬКОЇ СИСТЕМИ

3.1 Практична реалізація об’єкта проєктування

У межах даної кваліфікаційної роботи було реалізовано повноцінну програмну систему, призначену для імітації, передачі, обробки та візуалізації телеметричних даних повітряного руху в умовах моделювання авіадиспетчерського середовища. Система побудована з урахуванням сучасних архітектурних підходів і реалізована як мікросервісне рішення з використанням технологій Java, Spring Boot, Apache Kafka, WebSocket та Nuxt.js. Основною функціональною метою створеного програмного забезпечення є забезпечення безперервного потоку телеметричних повідомлень між окремими сервісами, їх агрегація, перевірка достовірності, трансформація у придатний формат та подальша передача до клієнтського інтерфейсу в режимі реального часу. Така постановка задачі дозволила сконцентрувати увагу не лише на технічних аспектах розробки, але й на правильному проектуванні взаємодії між мікросервісами в умовах підвищеного навантаження та часової критичності [9].

З точки зору реалізації, першим ключовим компонентом стала серверна частина, що відповідає за генерацію або прийом телеметричних повідомлень, умовно названа сервісом аеропорту або Office-сервісом. У його завдання входить формування телеметричних даних, які включають координати повітряного судна, швидкість його руху, напрям, висоту та інші параметри, що є типовими для систем авіаційного спостереження. Відповідно до режиму роботи система може або генерувати тестові пакети даних самостійно, використовуючи планувальник завдань Spring з анотацією @Scheduled (ліст. 3.1), або ж приймати їх із зовнішніх джерел – за умови інтеграції з реальними системами [10].

Лістинг 3.1 – Планувальник генерації телеметричних повідомлень

```
@Service
public class TelemetryScheduler {

    private final KafkaTemplate<String, TelemetryData> kafkaTemplate;
    private final TelemetryGenerator telemetryGenerator;

    @Autowired
    public TelemetryScheduler(KafkaTemplate<String, TelemetryData>
kafkaTemplate,
                              TelemetryGenerator telemetryGenerator) {
        this.kafkaTemplate = kafkaTemplate;
        this.telemetryGenerator = telemetryGenerator;
    }

    @Scheduled(fixedRate = 2000)
    public void sendMockTelemetry() {
        TelemetryData data = telemetryGenerator.generateMockData();
        kafkaTemplate.send("airport-topic", data.getPlaneId(), data);
        System.out.println("Mock telemetry sent for plane: " +
data.getPlaneId());
    }
}
```

кінець лістингу 3.1

Формовані повідомлення кодуються у форматі JSON, після чого відправляються до відповідних Kafka-топіків (ліст. 3.2), які відіграють роль транспортного каналу для інших сервісів системи.

Лістинг 3.2 – Відправлення повідомлення до Kafka з форматом JSON

```
@Configuration
public class KafkaProducerConfig {

    @Bean
    public ProducerFactory<String, TelemetryData> producerFactory() {
        Map<String, Object> configProps = new HashMap<>();
        configProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
"localhost:9092");
        configProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        configProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
JsonSerializer.class);
        configProps.put(JsonSerializer.ADD_TYPE_INFO_HEADERS, false);
// Опціонально
```

```
        return new DefaultKafkaProducerFactory<>(configProps);
    }

    /**
     * Створює KafkaTemplate – об’єкт для зручного надсилання
повідомлень.
     * @return KafkaTemplate для надсилання об’єктів типу
TelemetryData.
     */
    @Bean
    public KafkaTemplate<String, TelemetryData> kafkaTemplate() {
        return new KafkaTemplate<>(producerFactory());
    }
}
```

кінець лістингу 3.2

Таким чином, цей сервіс виконує роль джерела телеметрії, ініціатора даних для всієї архітектури.

Другим фундаментальним модулем є сервіс, що умовно названий сервісом літаків або Ship-сервісом. Його логіка полягає у споживанні повідомлень з Kafka, обробці отриманої інформації, верифікації коректності координат, а також у відстеженні логіки маршрутизації повітряного судна. Після підписки на відповідні Kafka-топіки за допомогою анотації @KafkaListener (ліст. 3.3), сервіс отримує потік даних, кожен елемент якого проходить процедуру нормалізації [11].

Лістинг 3.3 – Слухач Kafka у сервісі літаків

```
@Component
public class PlaneTelemetryListener {

    private final PlaneService planeService;

    @Autowired
    public PlaneTelemetryListener(PlaneService planeService) {
        this.planeService = planeService;
    }
    @KafkaListener(topics = "airport-topic", groupId = "plane-
service")
    public void listen(ConsumerRecord<String, PlaneTelemetry> record)
{
        String planeId = record.key();
        PlaneTelemetry telemetry = record.value();
```

```
        System.out.println("Received telemetry for plane: " +
planeId);

        planeService.processTelemetry(planeId, telemetry);
    }
}
```

кінець лістингу 3.3

Сюди входить перевірка діапазонів координат, фільтрація повідомлень з неповними або некоректними значеннями, а також запис повідомлень у тимчасовий буфер для подальшої агрегації. Крім того, реалізована логіка виявлення аномальних змін у курсі чи швидкості, які можуть свідчити про відхилення від маршруту або позаштатну ситуацію. Оброблені повідомлення після цього переадресовуються через Kafka до наступного сервісу (ліст. 3.4), таким чином цей модуль виконує роль обробника, валідаційного фільтра та посередника між джерелом і кінцевим споживачем.

Лістинг 3.4 – Переадресація обробленого повідомлення в інший Kafka-топік

```
public void handle(TelemetryData data) {
    if (validator.isValid(data)) {
        kafkaTemplate.send("processed-topic", data);
    }
}
```

кінець лістингу 3.4

Третім ключовим сервісом у межах розробленої архітектури є WebSocket-сервіс, що забезпечує стійке двостороннє з’єднання з клієнтською частиною. Він не займається складною логікою обробки, а виконує роль рефлектора – сервісу, що ретранслює вже підготовлені повідомлення з внутрішніх топіків Kafka безпосередньо на клієнт. Відмінністю такого підходу від звичайного HTTP-запиту є постійне підтримання з’єднання з браузером, що дозволяє не втрачати час на ініціалізацію запитів, а також надсилати повідомлення миттєво після їх отримання. На рівні коду було реалізовано автоматичне встановлення WebSocket-сесії для кожного клієнта (ліст. 3.5),

приєднаного до системи, з подальшою маршрутизацією повідомлень за сесіями. Такий підхід гарантує низьку затримку передачі даних та високу частоту оновлення візуалізованої інформації на клієнтському інтерфейсі.

Лістинг 3.5 – Ініціалізація WebSocket-сесії та надсилання повідомлень

```
@OnOpen
public void onOpen(Session session) {
    sessions.add(session);
}

public void broadcast(String message) {
    for (Session s : sessions) {
        s.getAsyncRemote().sendText(message);
    }
}
```

кінець лістингу 3.5

Зі сторони кінцевого користувача доступ до оброблених телеметричних даних забезпечується за допомогою клієнтського застосунку, який реалізовано з використанням сучасного JavaScript-фреймворку Nuxt.js [12]. Цей фронтенд-компонент відіграє ключову роль у формуванні інтерактивного інтерфейсу взаємодії між диспетчером та інформаційною системою. Його головним призначенням є візуалізація повітряного простору, що охоплює графічне представлення поточного положення повітряних суден, їх маршрутів, а також основних елементів навігаційної інфраструктури.

Крім того, у межах інтерфейсу реалізовано можливість відображення статусу виконання польотів, що включає графічне й текстове представлення маршруту літака, його поточного стану, швидкості, висоти, напряму руху тощо. Візуальне представлення цього функціоналу наведено на рисунку 3.1, який демонструє логіку маршруту та послідовність передачі даних про літак до відповідних офісів керування.

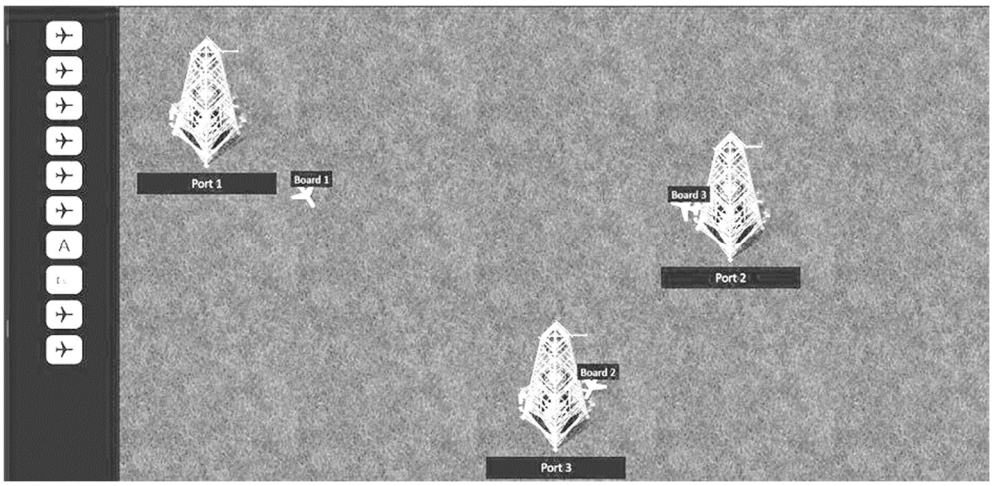

Рисунок 3.1 – Виконання маршрутів літаками

Для забезпечення інтерактивності використовуються механізми реактивного оновлення DOM, а також підписка на WebSocket-канал (ліст. 3.6).

Лістинг 3.6 – Підписка Nuxt.js клієнта на WebSocket

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements
WebSocketMessageBrokerConfigurer {

    @Override
    public void configureMessageBroker(MessageBrokerRegistry config) {
        config.enableSimpleBroker("/topic");
        config.setApplicationDestinationPrefixes("/app");
    }

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry)
{
        registry.addEndpoint("/ws-plane")
                .setAllowedOrigins("*")
                .withSockJS();
    }
}
```

кінець лістингу 3.6

У результаті кінцевий користувач – авіадиспетчер або оператор – бачить на екрані постійно оновлювану карту повітряного руху із зазначенням ключових параметрів. Окрім цього, в інтерфейс інтегровано додаткові

інформаційні панелі, що відображають загальну кількість літаків у повітрі, лог подій, статус з’єднання та інші службові відомості.

Усі сервіси було реалізовано з дотриманням принципів об’єктно-орієнтованого програмування та шаблонів проєктування. Кожен сервіс містить власні конфігураційні класи, сервіси бізнес-логіки, обробники подій, DTO-моделі (ліст. 3.7), що дозволяє чітко відділити відповідальність між компонентами. В якості засобу ін’єкції залежностей використано стандартні анотації Spring @Component, @Service, @Autowired, що забезпечує слабке зв’язування між модулями і спрощує модульне тестування.

Лістинг 3.7 – DTO-модель для передачі телеметричних даних

```
public class TelemetryData {
    private String planeId;
    private double latitude;
    private double longitude;
    private double altitude;
}
```

кінець лістингу 3.7

Також передбачено окрему конфігурацію Kafka-продюсерів і консьюмерів (ліст. 3.8), що дозволяє гнучко змінювати топіки або налаштування передачі повідомлень без втручання у бізнес-логіку сервісів.

Лістинг 3.8 – Конфігурація Kafka-консьюмера

```
@Bean
public ConsumerFactory<String, String> consumerFactory() {
    Map<String, Object> config = new HashMap<>();
    config.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
"localhost:9092");
    config.put(ConsumerConfig.GROUP_ID_CONFIG, "ship-service");
    return new DefaultKafkaConsumerFactory<>(config);
}
```

кінець лістингу 3.8

Загалом, система має повністю контейнеризовану структуру: кожен мікросервіс збирається в окремий Docker-образ, а їх спільне розгортання здійснюється через docker-compose (ліст. 3.9), який також включає сервіси Kafka, Zookeeper і frontend [13].

Лістинг 3.9 – Docker-compose.yml для всіх сервісів

```
version: ‘2’
services:
    zookeeper:
        image: confluentinc/cp-zookeeper:latest
        environment:
            ZOOKEEPER_CLIENT_PORT: 2181
            ZOOKEEPER_TICK_TIME: 2000
        ports:
            - 2181:2181

    kafka:
        image: confluentinc/cp-kafka:latest
        depends_on:
            - zookeeper
        ports:
            - 29092:29092
        environment:
            KAFKA_BROKER_ID: 1
            KAFKA_ZOOKEEPER_CONNECT: ‘zookeeper:2181’
            KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://localhost:29092
            KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
            KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
```

кінець лістингу 3.9

Це забезпечує легке масштабування, можливість розгортання як у локальному середовищі, так і на хмарній інфраструктурі. Крім того, використання брокера Kafka дозволяє реалізовувати механізми повторної доставки повідомлень, зберігання подій та їх відтворення у разі відмови.

Таким чином, у межах розробки було створено повністю функціональний набір сервісів, кожен з яких виконує окрему роль у загальній логіці системи. Їхня взаємодія забезпечує наскрізний потік даних від симуляції телеметрії до її візуалізації, що демонструє практичну реалізацію принципів

сучасної мікросервісної архітектури, реактивного програмування та передачі повідомлень у реальному часі.

3.2 Тестування та налагодження інформаційно-комп'ютерної системи

У процесі розробки складних розподілених інформаційних систем, зокрема таких, що працюють у режимі реального часу й мають справу з великою кількістю асинхронних подій, тестування виступає не просто етапом перевірки коректності роботи програмного забезпечення, а критично важливою складовою всього життєвого циклу розробки. У випадку розробленого програмного забезпечення для авіадиспетчерської системи, тестування відігравало центральну роль на всіх етапах – від модульного до інтеграційного та навантажувального рівнів. Це обумовлено необхідністю гарантувати стійкість, надійність, продуктивність і своєчасну реакцію на події, що є вкрай важливими у контексті повітряного руху.

Початково, на ранніх етапах розробки, було реалізовано тестування окремих мікросервісів. Оскільки вся система побудована за мікросервісною архітектурою, кожен сервіс розгортався, тестувався і відлагоджувався ізольовано. Для цього використовувалося вбудоване тестове середовище Spring Boot, яке дозволяє запускати окремі компоненти без необхідності піднімати всю систему. Зокрема, перевірялась логіка обробки телеметричних повідомлень, парсинг даних, трансформація у формат, зручний для подальшої передачі через Kafka, а також їх передача у вигляді подій.

Особливу увагу було приділено тестуванню взаємодії між сервісами через брокер Kafka. У реальних умовах система має справу з десятками, а інколи й сотнями подій у секунду. Кожна подія передається між сервісами у вигляді Kafka-повідомлення. Тому важливо було переконатися, що сервіс-продюсер (який публікує події, наприклад телеметрію) не тільки правильно формує повідомлення, але й що консюмер (сервіс, який приймає події) вчасно

і коректно їх обробляє. Для цього було створено тестовий пайплайн, де генератор подій імітував надходження даних про повітряні судна з певною частотою. Сервіси обробки й візуалізації мали вчасно реагувати на ці події, а результат обробки – відображатись у реальному часі на стороні клієнта.

Тестування WebSocket-з’єднання також мало окрему вагу. Через нього реалізовано безперервну трансляцію даних до клієнтської частини – інтерфейсу авіадиспетчера. У процесі тестування перевірялась стійкість з’єднання до обривів, час реакції при надходженні нових повідомлень, затримки при великому потоці даних. Була змодельована ситуація, коли клієнт втрачає зв’язок із сервером, і перевірено, як швидко та без втрати даних відновлюється сесія.

Налагодження сервісів проводилося через консольний логінг, використовуючи систему логів, які виводилися для кожного етапу обробки події: надходження повідомлення, його парсинг, публікація в Kafka, отримання іншими сервісами, відображення на фронтенді. Цей підхід дозволив швидко виявляти і локалізовувати проблеми – наприклад, якщо певне повідомлення некоректно формувалося або загублювалося під час трансляції.

Задля забезпечення контролю над продуктивністю системи проводились також тестування під навантаженням. Використовуючи емулятор потоків даних (симулятор телеметрії), система отримувала потік з тисяч повідомлень у хвилину. В ході тестування спостерігалося використання ресурсів (CPU, RAM), швидкість обробки, затримки при публікації в Kafka та реакція клієнта. Ці тести дозволили скоригувати параметри Kafka-брокера (розмір батчів, час затримки обробки, розподіл партицій), а також внутрішню обробку у споживачах, щоб уникнути чергування та затримок.

Особливу увагу приділено перевірці масштабованості системи. Оскільки кожен мікросервіс розгортається окремо, було протестовано запуск декількох інстансів одного і того ж сервісу. Наприклад, у ситуації, коли кількість подій зростає, можливе горизонтальне масштабування сервісу-споживача Kafka. Усі

сервіси були протестовані на предмет здатності обробляти паралельні потоки подій, що було реалізовано завдяки використанню Kafka groupId для балансування навантаження [14].

Системні вимоги до роботи застосунку формувалися з урахуванням розміру потоків даних та необхідної швидкодії. Для серверної частини мінімальні вимоги передбачають 2-ядерний процесор, 4 ГБ оперативної пам’яті, Java 17 і встановлене середовище Docker для розгортання Kafka-брокера. Рекомендовані параметри для продуктивного використання – 4 ядра, 8 ГБ ОЗП, SSD-диск, розділення сервісів на окремі контейнери. Для клієнтської частини достатньо сучасного браузера з підтримкою WebSocket, бажано Google Chrome або Firefox останніх версій.

Завдяки багатоетапному тестуванню, включно з інтеграційним та функціональним, вдалось досягти стабільної роботи системи навіть при високому навантаженні, забезпечити відсутність втрат повідомлень, затримок та критичних помилок. Цей етап став завершальним перед розгортанням системи у тестовому середовищі та демонстрацією її можливостей.

3.3 Розгортання інформаційно-комп’ютерної системи

Процес розгортання інформаційно-комп’ютерної системи, побудованої за принципами мікросервісної архітектури, є одним з ключових етапів життєвого циклу програмного забезпечення. Саме на цьому етапі створений програмний продукт переходить із локального середовища розробки до умов, наближених до експлуатаційних. У межах цієї кваліфікаційної роботи розгортання системи здійснювалось у тестовому серверному середовищі з метою перевірки її стабільності, масштабованості та здатності до роботи у режимі реального часу.

З урахуванням того, що система складається з декількох незалежних мікросервісів, було прийнято рішення розгортати кожен сервіс окремо у вигляді ізольованого контейнера. Для цього використано платформу Docker,

яка забезпечує ізоляцію середовища, зручність конфігурації, масштабованість і мобільність між різними хостами. Контейнеризація дозволила уникнути проблем сумісності між різними версіями середовищ виконання, бібліотек та системних залежностей.

На першому етапі було створено Dockerfile для кожного сервісу, в якому описано послідовність інструкцій для збирання контейнера: вибір базового образу, копіювання артефактів (JAR-файлів), встановлення змінних середовища, конфігураційних файлів, а також команд для запуску. Після цього всі сервіси об’єднано в єдину мережу через файл docker-compose.yml, що дозволило забезпечити їх взаємозв’язок без необхідності вручну налаштовувати маршрутизацію.

Kafka, як центральний компонент системи обміну повідомленнями, також було розгорнуто в окремому контейнері за допомогою образів з офіційного репозиторію. Для спрощення конфігурації використано Confluent-платформу, яка містить не лише брокер Kafka, але й ZooKeeper, Schema Registry, KSQL тощо. Це дозволило легко налаштувати базовий кластер Kafka з можливістю масштабування кількості брокерів та партицій у майбутньому.

Кожен із сервісів було протестовано на підключення до Kafka, перевірено коректність відправки та отримання повідомлень, налаштовано системи логування та моніторингу. Для спрощення спостереження за станом системи інтегровано Kafka UI – інтерфейс, який дозволяє візуалізувати повідомлення, що проходять крізь топіки, та швидко ідентифікувати збої у процесі обробки.

Фронтенд-додаток, розроблений за допомогою Nuxt.js, було зібрано у production-режимі за допомогою команди npm run generate. Для WebSocket-з’єднання використано вбудовані можливості Nuxt через плагіни на основі socket.io-client, що дозволило підтримувати безперервний зв’язок із сервером для відображення телеметричних даних у режимі реального часу.

Налагодження конфігураційних параметрів (зокрема, URL-адрес сервісів, портів, Kafka-топіків) здійснювалось за допомогою змінних

середовища, які передаються в контейнер при запуску. Це забезпечило гнучкість і повторюваність процесу розгортання – система може бути легко перенесена на інший сервер без зміни самого коду.

Важливо, що вся система може бути розгорнута однією командою завдяки docker-compose, що особливо корисно при розробці, тестуванні або демонстрації. Команда docker-compose up --build дозволяє автоматично зібрати всі сервіси, запустити Kafka-кластер, підключити всі компоненти та забезпечити повну працездатність системи.

Окрему увагу приділено відкриттю портів у системі для зовнішнього доступу: Kafka було налаштовано з проксуванням через внутрішню мережу, тоді як фронтенд – з доступом через публічний порт 80. Для безперебійної роботи WebSocket-з'єднання передбачено захист від переривань та повторні з'єднання у разі втрати зв'язку.

Таким чином, у межах етапу розгортання було реалізовано повний цикл підготовки системи до запуску в тестовому середовищі: створення контейнерів, налаштування мережі, запуск Kafka, підключення мікросервісів, деплой Nuxt-додатку та забезпечення стабільного з'єднання з усіма компонентами. Це доводить можливість подальшого розширення системи на рівень промислової експлуатації за умови мінімальних змін у конфігурації.

3.4 Формування журналу подій

У рамках розробки програмного забезпечення для авіадиспетчерської особливу увагу приділено питанням ведення журналу подій, який є важливим елементом моніторингу, відстеження та аналізу роботи системи в реальному часі. Формування такого журналу забезпечує прозорість функціонування кожного мікросервісу, а також можливість оперативного виявлення та усунення несправностей.

В умовах мікросервісної архітектури, де кожен сервіс виконує свою специфічну роль і взаємодіє з іншими компонентами через Kafka, особливо

важливо мати централізований підхід до логування. Для цього у розробленій системі застосовано концепцію агрегованого журналу подій, що формується шляхом збору логів від усіх сервісів у єдину централізовану платформу. Приклад логування сервісу Ship відображений в рисунку 3.2.

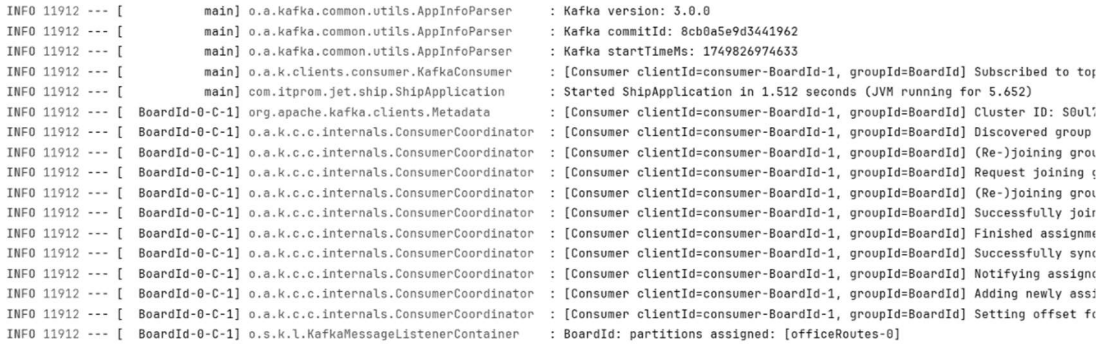

Рисунок 3.2 – Консольне логування сервісу Ship після його запуску

Кожен із сервісів містить власні механізми логування, що реалізовані за допомогою вбудованих бібліотек Spring Boot. Логи включають інформацію про запуск і зупинку сервісів, відправlення та отримання повідомлень Kafka, помилки, попередження, а також специфічні події, пов’язані з обробкою авіаційних даних. Для полегшення аналізу всі повідомлення структуровані з використанням формату JSON, що дозволяє ефективно обробляти записи у системах збору логів.

У клієнтській частині інтерфейсу реалізовано систему візуального маркування подій. Уздовж лівого краю розміщено графічні індикатори, які відображають факт обробки певної події в системі. Кожна іконка репрезентує подію певного типу: наприклад, зображення літака позначає зміну або ініціацію маршруту, а літера «A» – подія завершення маршруту, літак повідомлює свій стан і кількість літаків в ньому.

При наведенні курсора на піктограму відкривається інформаційна підказка (tooltip), яка містить структуровані дані у форматі JSON. Ці дані включають тип події (type), джерело її генерації (source), інформацію про маршрут (path) тощо.

Такий підхід поєднує логування з візуалізацією, дозволяючи оператору швидко отримати діагностичну інформацію без необхідності звернення до консолі або журналу подій. Це значно спрощує процес налагодження та моніторингу симуляції. Приклад відображення логування в UI, наведений в рисунку 3.3.

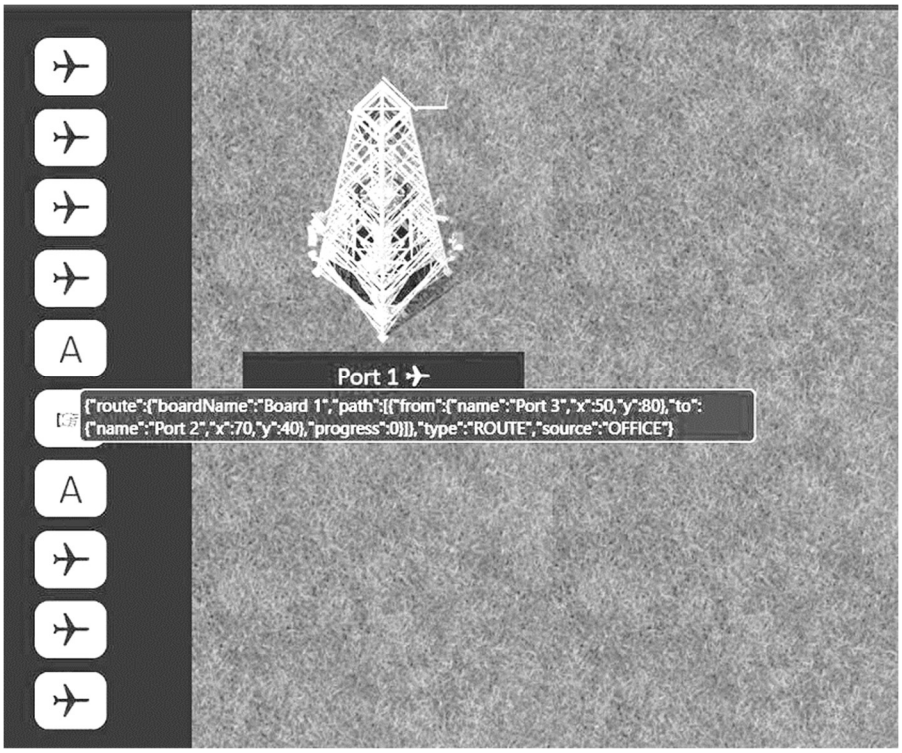

Рисунок 3.3 – Відображення логів подій у графічному інтерфейсі користувача

Взаємодія з Kafka, яка є основним брокером повідомлень у системі, також ретельно контролюється. Кожен відправлений або отриманий Kafka-меседж супроводжується записом у журнал подій із зазначенням часу, ідентифікатора повідомлення, типу події, а також імені топіка. Це дозволяє відтворити повну картину обміну даними в системі і встановити, на якому етапі могло виникнути відхилення або помилка.

Для централізації логів використано інструменти типу ELK Stack (Elasticsearch, Logstash, Kibana) або альтернативні рішення для збору, індексації та візуалізації журналів. Хоча впровадження такого стека не було обов’язковим у межах даної роботи, архітектура системи спроектована з

урахуванням можливості подальшої інтеграції з подібними системами моніторингу.

У разі виникнення критичних помилок або аномалій система має можливість відправляти оповіщення адміністратору через спеціальні сервіси оповіщення (наприклад, email або webhook), що істотно підвищує оперативність реагування на проблеми.

Особливу увагу також приділено захисту та збереженню журналів. Оскільки авіадиспетчерська система оперує конфіденційною інформацією, ведення журналу подій організовано з урахуванням вимог до безпеки: використання доступу за ролями, шифрування архівів логів, а також періодичне архівування з метою збереження історичних даних.

Таким чином, формування журналу подій у створеній системі не лише забезпечує високу прозорість і контроль над роботою кожного компонента, а й створює фундамент для масштабованого та безпечного функціонування системи в умовах реального часу. Це є важливим кроком на шляху до подальшого впровадження розробленого програмного забезпечення у виробниче середовище.

3.5 Підтримка проєкту та використання системи контролю версій GitHub

В процесі розробки сучасних програмних систем особливе значення має організація контролю версій коду та автоматизація процесів розгортання і тестування. У даному проєкті для цих цілей було обрано платформу GitHub, яка поєднує в собі функції системи контролю версій, хмарного сховища коду, а також потужних інструментів для автоматизації робочих процесів за допомогою GitHub Actions [15].

Система контролю версій Git дозволяє відслідковувати всі зміни в коді, забезпечувати безпечну роботу з кількома паралельними гілками розробки, що особливо важливо для підтримки стабільності великого проєкту з кількома

мікросервісами. Основна структура репозиторію організована у вигляді двох основних гілок – main і develop. Гілка main містить стабільний код, що готовий до розгортання, тоді як develop слугує для інтеграції нових функцій та змін. Така організація дозволяє уникнути конфліктів та забезпечити контроль якості коду через процес pull request, де кожен новий внесок проходить обов'язкове рев'ю іншими членами команди. Приклад поступового додавання процесу розробки наведено в рисунку 3.4.

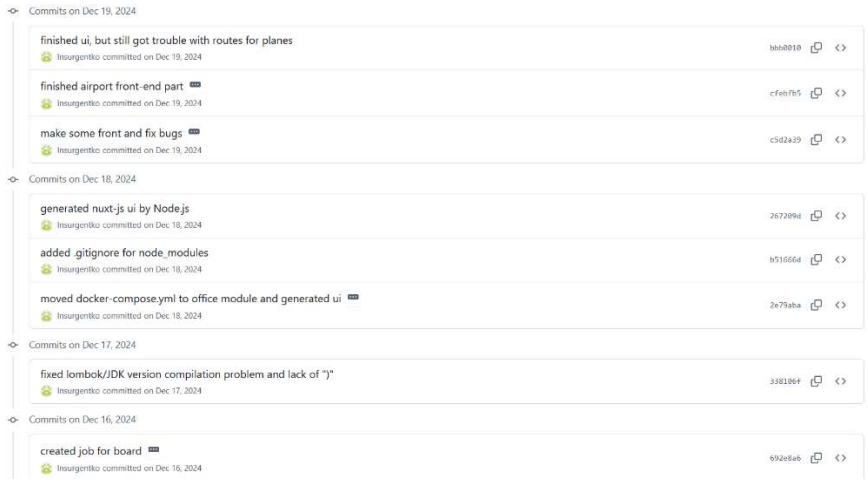

Рисунок 3.4 – Поступове додавання нововведень через контроль версій GitHub

Для покращення якості коду і прискорення циклу розробки застосовуються GitHub Actions – вбудована система CI/CD (Continuous Integration / Continuous Deployment). За допомогою YAML-конфігурацій описані робочі процеси, які автоматично запускаються при кожному коміті або створенні pull request. Ці процеси включають збірку проєкту, запуск модульних тестів, статичний аналіз коду на наявність помилок і потенційних проблем, а також формування звітів. Завдяки цьому розробники мають змогу виявляти проблеми на ранніх етапах, що значно підвищує стабільність і надійність програмного продукту.

Наприклад, один з основних робочих процесів GitHub Actions у проєкті виглядає наступним чином: після кожного пушу в гілку develop запускається

збірка Java-сервісів з використанням Maven, потім автоматично запускаються юніт-тести для перевірки коректності роботи бізнес-логіки. У разі виявлення помилок, розробник отримує повідомлення у вигляді коментаря у pull request, що дозволяє оперативно виправити проблеми. Такий підхід дозволяє підтримувати високу якість коду та уникати помилок на продуктивному середовищі.

Для розгортання сервісів використовується Docker – система контейнеризації, яка ізолює програмне середовище кожного мікросервісу, зберігаючи його залежності і конфігурацію у вигляді окремих образів. Ці Docker-образи збираються також автоматично за допомогою GitHub Actions і зберігаються у GitHub Container Registry, що спрощує керування версіями і дозволяє швидко оновлювати сервіси на сервері. Такий підхід забезпечує однакове середовище для розробки, тестування та продакшену, зводячи до мінімуму проблеми сумісності та «працює у мене» ("works on my machine").

Окрему увагу приділено детальній документації, яка також зберігається у репозиторії. Вона містить опис структури проекту, інструкції по клонуванню репозиторію, запуску контейнерів та налаштуванню локального середовища. Документація написана у Markdown, що забезпечує її зручне читання як у GitHub, так і локально. Для веб-інтерфейсу описані кроки з встановлення залежностей Nuxt.js, запуску локального сервера і збірки проекту для продакшену. Документація допомагає новим розробникам швидко включитися у роботу і знизити час на адаптацію.

GitHub також дозволяє формувати релізи – стабільні версії коду, що позначаються тегами. Кожен реліз супроводжується описом змін (changelog), що полегшує моніторинг еволюції продукту. Такий підхід сприяє кращому управлінню версіями і плануванню подальших оновлень. У разі виявлення критичних багів є можливість швидко відкотити зміни до попередньої стабільної версії, що дуже важливо для програмних продуктів з високими вимогами до надійності, таких як авіадиспетчерські системи.

Використання GitHub значно покращило командну взаємодію: завдяки централізованому репозиторію всі учасники мають доступ до актуального коду, можуть залишати коментарі, брати участь у рев'ю змін та швидко реагувати на помилки. Автоматизація процесів зменшує кількість рутинної роботи і дозволяє зосередитися на розробці нових функціональних можливостей.

Підсумовуючи, використання GitHub як платформи для зберігання коду, контролю версій, автоматизації тестування і розгортання стало невід'ємною частиною процесу розробки даного проєкту. Це дозволило забезпечити високу якість, стабільність і гнучкість програмного забезпечення, що є необхідним для ефективної роботи системи авіадиспетчерського управління у реальному часі. Завдяки такому підходу було досягнуто високого рівня організації робочого процесу, що позитивно впливає на подальший розвиток і підтримку програмного продукту.

Висновки до розділу 3

Розділ 3 присвячений практичній реалізації програмного забезпечення для авіадиспетчерської системи, що базується на мікросервісній архітектурі з використанням сучасних технологій Spring Boot, Kafka, WebSocket та Nuxt.js. У результаті виконано комплексну розробку, що охоплює створення сервісів, налаштування їх взаємодії через брокер повідомлень Kafka, а також реалізацію механізмів обробки та передачі даних у реальному часі.

В процесі розробки було детально опрацьовано основні функціональні компоненти системи, реалізовано бізнес-логіку окремих мікросервісів, а також інтеграцію з інтерфейсом користувача на базі Nuxt.js, що забезпечує гнучке та швидке відображення актуальної інформації про повітряний рух. Застосування Spring Boot надало можливість створити незалежні, легко масштабовані сервіси з високим рівнем підтримки, а Kafka стала центральним елементом

системи, що гарантує надійну доставку повідомлень і організовує асинхронну взаємодію між компонентами.

Значна увага приділялась якості коду та застосуванню ефективних методів тестування і налагодження. В процесі розробки було проведено модульне та інтеграційне тестування, що забезпечило стабільність і передбачуваність роботи системи навіть у випадках високих навантажень і великих потоків даних. Виявлені недоліки та помилки були своєчасно усунуті, що свідчить про високий рівень організації процесу розробки.

Особливим внеском у стабільність системи стало застосування механізмів формування журналу подій, які дозволяють відслідковувати стан кожного сервісу, фіксувати ключові операції, повідомлення Kafka та потенційні аномалії. Централізований підхід до логування з використанням структурованих форматів полегшує подальший аналіз і моніторинг, створюючи надійний інструмент для підтримки експлуатації.

Враховуючи особливості проєкту, було розроблено ефективні стратегії автоматизації запуску і управління сервісами, що значно спрощує розгортання і оновлення програмного комплексу. Конфігурація компонентів побудована так, щоб забезпечувати масштабованість та легкість інтеграції з іншими системами.

Таким чином, реалізована система відповідає поставленим вимогам і завданням, які полягали у створенні універсального програмного забезпечення для авіадиспетчерської, здатного обробляти та відображати дані у реальному часі. Практична реалізація підтвердила, що обрана архітектура та технологічний стек дозволяють досягнути високої продуктивності, надійності та гнучкості.

Отже, розроблена інформаційно-комп’ютерна система є надійним прототипом, який може бути використаний як у навчальних цілях, так і служити основою для подальшого промислового впровадження. Вона задовольняє сучасні вимоги авіаційної галузі до обробки великих потоків

даних у реальному часі і має потенціал для масштабування та розширення функціоналу з урахуванням майбутніх потреб користувачів.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи на тему «Розробка програмного забезпечення для авіадиспетчерської з використанням Spring, Kafka, WebSocket та Nuxt.js» досягнуто основної мети – створено сучасне програмне забезпечення, що забезпечує надійну, масштабовану та ефективну обробку і візуалізацію телеметричних даних у режимі реального часу.

На початковому етапі проведено дослідження сучасних підходів до архітектури програмних систем у сфері повітряного руху. Здійснено аналіз предметної області, розглянуто типові архітектурні рішення та проблематику авіадиспетчерських систем, що дало змогу визначити вимоги до функціональності та гнучкості майбутнього рішення.

У процесі технічного обґрунтування вибрано оптимальний стек технологій: Spring Boot, Apache Kafka, WebSocket та Nuxt.js. Вибір продиктовано необхідністю забезпечення високої продуктивності, низької затримки обміну даними та підтримки двосторонньої взаємодії між компонентами системи.

Архітектура програмного забезпечення спроєктована відповідно до мікросервісного підходу. Це дало змогу розділити функціональність на автономні модулі, що легко масштабуються, тестуються й підтримуються незалежно один від одного.

Розроблено серверні сервіси за допомогою Spring Boot, реалізовано обмін повідомленнями через Kafka та організовано двосторонню комунікацію між клієнтом і сервером за допомогою WebSocket. Такий підхід дозволив досягти стабільної роботи системи в умовах реального часу та динамічного навантаження.

Клієнтський інтерфейс створено з використанням Nuxt.js. Він забезпечує інтуїтивну візуалізацію повітряного простору та зручну взаємодію диспетчера з інформаційною системою.

Усі ключові функції: реєстрація телеметрії, передача повідомлень між сервісами, візуалізація та інтерактивна взаємодія реалізовані відповідно до визначених вимог. Система працює з мінімальною затримкою, надаючи актуальні дані у зручному форматі.

Особливу увагу приділено тестуванню. Проведено перевірку як окремих компонентів, так і всієї системи загалом із застосуванням юніт-, інтеграційних та ручних тестів, що забезпечило впевненість у стабільності та надійності програмного забезпечення.

Наприкінці виконано загальний аналіз ефективності запропонованого рішення. Отримані результати підтверджують успішну реалізацію усіх поставлених завдань, демонструють придатність системи до подальшого масштабування та інтеграції в навчальне або виробниче середовище.

Таким чином, у межах кваліфікаційної роботи реалізовано повний цикл розробки сучасної авіадиспетчерської системи – від аналізу предметної області до створення та тестування програмного комплексу, що відповідає актуальним вимогам ринку програмного забезпечення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Richards M., Ford N. Fundamentals of Software Architecture: An Engineering Approach. O’Reilly Media, 2020. 432 p.

2. O'Reilly P. Microservices Patterns: With Examples in Java. Manning Publications, 2020. 520 p.

3. Banning A. Real-Time Analytics: Techniques to Analyze and Visualize Streaming Data. O’Reilly Media, 2021. 320 p.

4. Walls C. Spring in Action. 6th ed. Manning Publications, 2022. 520 p.

5. Fowler M. Refactoring: Improving the Design of Existing Code. 2nd ed. Addison-Wesley, 2020. 448 p.

6. Apache Kafka. Apache Kafka Documentation – Concepts. URL: https://kafka.apache.org/documentation/#intro_concepts (дата звернення: 08.04.2025).

7. Fette I., Melnikov A. The WebSocket Protocol (RFC 6455). Internet Engineering Task Force. URL: https://datatracker.ietf.org/doc/html/rfc6455 (дата звернення: 10.04.2025).

8. Vue.js Documentation. URL: https://vuejs.org/guide/introduction.html (дата звернення: 11.04.2025).

9. Richards M. Software Architecture: The Hard Parts. O’Reilly Media, 2021. 336 p.

10. Spring Boot. Spring Boot Reference Documentation. URL: https://docs.spring.io/spring-boot/docs/current/reference/html/ (дата звернення: 08.05.2025).

11. Glover B., Cebula B., Shapira G. Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale. 2nd ed. O’Reilly Media, 2021. 450 p.

12. Nuxt.js. Nuxt 3 Documentation. URL: https://nuxt.com/docs/getting-started/introduction (дата звернення: 15.05.2025).

13. Turnbull J. The Docker Book: Containerization is the New Virtualization. Independently published, 2021. 348 p.

14. Glover B., Cebula B. Kafka Streams in Action. Manning Publications, 2021. 360 p.

15. Cogswell B. Learning GitHub Actions. O’Reilly Media, 2020. 250 p.