Міністерство освіти і науки України
Криворізький національний університет
Факультет інформаційних технологій
Кафедра автоматизації, комп’ютерних наук і технологій

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття ступеня вищої освіти – бакалавр

за освітньо-професійною програмою

«Комп’ютерні науки»

зі спеціальності

122 – Комп’ютерні науки

тема роботи:

«Система обробки заявок для аварійних служб на основі мікросервісної архітектури та месенджерів»

Виконав студент гр. КН-22-2 _____ Рабенюк Д.А.

Керівник _____ Гриценко А.М.

Нормоконтроль _____ Маринич І. А.

Завідувач кафедри _____ Рубан С. А.

Кривий Ріг – 2026

АНОТАЦІЯ

Рабенюк Д.А. Система обробки заявок для аварійних служб на основі мікросервісної архітектури та месенджерів: кваліфікаційна робота бакалавра : 122 – Комп’ютерні науки. Кривий Ріг. Криворізький національний університет, 2026. 81 с.

У роботі розглянуто проєктування та реалізацію інформаційної системи для прийому, реєстрації, маршрутизації й контролю виконання аварійних заявок. Система автоматизує обробкЗ.у звернень, зменшує навантаження на операторів і забезпечує своєчасне інформування заявників.

Метою роботи є створення системи на основі мікросервісної архітектури та месенджерів, яка підтримує приймання звернень, класифікацію, визначення пріоритету, призначення виконавців, зміну статусів і надсилання повідомлень.

Практичне значення полягає у створенні працездатного прототипу для аварійних, комунальних і технічних служб. Рішення підтримує Telegram, рольовий доступ, асинхронний обмін повідомленнями та контейнерне розгортання. Система також забезпечує збереження історії опрацювання заявок, передачу фотографій і геолокації та контроль основних етапів їх виконання.

У першому розділі проаналізовано предметну область, існуючі системи та архітектурні підходи. У другому розділі спроєктовано архітектуру, бізнес-процес, базу даних, API, інтеграцію з Telegram і засоби захисту. У третьому розділі реалізовано мікросервіси, вебінтерфейс, Telegram-бот, обмін повідомленнями, тестування та розгортання.

Програмне рішення створено з використанням ASP.NET Core, PostgreSQL, RabbitMQ, Telegram Bot API і Docker. Проведене тестування підтвердило коректність основних функцій та взаємодії компонентів системи.

КЛЮЧОВІ СЛОВА: АВАРІЙНА ЗАЯВКА, МІКРОСЕРВІСИ, TELEGRAM-БОТ, REST API, RABBITMQ, POSTGRESQL, DOCKER.

ЗМІСТ

ВСТУП

Ефективність роботи аварійних і комунальних служб значною мірою залежить від швидкості прийому, обробки та виконання заявок населення. Пошкодження інженерних мереж, порушення водо-, тепло- або електропостачання та інші аварійні події потребують оперативного реагування. Затримка під час реєстрації звернення або його передачі відповідальній службі може призвести до збільшення збитків, погіршення умов проживання громадян і зниження рівня безпеки. В Україні приймання екстрених повідомлень передбачає використання відповідних телефонних номерів служб та єдиного номера 112 [1].

Телефонний зв’язок залишається поширеним способом подання аварійних заявок, проте під час масового надходження звернень лінії та оператори можуть бути перевантажені. Також існує ризик неповного або неточного запису адреси, опису проблеми й контактних даних. Заявник не завжди має можливість перевірити стан виконання звернення без повторного дзвінка. Це обґрунтовує необхідність застосування інформаційних систем, які забезпечують реєстрацію, класифікацію, маршрутизацію та контроль виконання заявок.

Одним із доступних цифрових каналів взаємодії із заявниками є месенджери. Через спеціалізованого бота користувач може вказати тип аварії, адресу, описати проблему, додати фотографію або передати геолокацію. Telegram Bot API надає HTTP-інтерфейс для створення ботів та підтримує обробку повідомлень, файлів, фотографій, координат і команд користувачів [2]. Такий канал дозволяє структурувати первинну інформацію, зменшити навантаження на операторів і автоматично повідомляти заявника про стан заявки.

Для реалізації системи доцільно застосувати мікросервісну архітектуру, яка передбачає поділ програмного рішення на автономні сервіси, кожен із яких реалізує окрему функціональну область і може незалежно розгортатися та масштабуватися [3]. У проєктованій системі такими компонентами можуть

бути сервіси заявок, користувачів, маршрутизації, сповіщень, бот-сервіс та аналітичний модуль. Взаємодія між ними може здійснюватися через API й асинхронні повідомлення, що зменшує зв’язаність компонентів і спрощує подальше розширення системи.

Актуальність кваліфікаційної роботи зумовлена необхідністю автоматизації прийому та супроводу аварійних заявок, підвищення оперативності реагування і забезпечення зворотного зв’язку із заявниками.

Метою роботи є проєктування та розробка системи обробки заявок для аварійних служб на основі мікросервісної архітектури та месенджерів, яка забезпечує прийом, збереження, класифікацію, маршрутизацію і контроль виконання звернень.

Об’єктом дослідження є процес прийому, реєстрації та супроводу заявок аварійними службами.

Предметом дослідження є архітектурні підходи та програмні засоби побудови системи обробки аварійних заявок з інтеграцією месенджера.

Для досягнення поставленої мети необхідно:

– проаналізувати предметну область та існуючі системи обробки звернень;

– дослідити можливості застосування месенджерів;

– обґрунтувати вибір мікросервісної архітектури та технологій;

– сформувати вимоги до системи;

– спроєктувати архітектуру, базу даних і взаємодію сервісів;

– реалізувати основні програмні компоненти;

– провести тестування та практичну апробацію системи.

Практична цінність роботи полягає у створенні прототипу, який може бути використаний як основа для автоматизації діяльності аварійних або комунальних служб. Запропоноване рішення має зменшити навантаження на операторів, підвищити прозорість обробки заявок та забезпечити оперативне інформування користувачів.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ ПІДХОДІВ ДО ОБРОБКИ ЗАЯВОК АВАРІЙНИХ СЛУЖБ

1.1 Аналіз предметної області аварійних служб

Аварійні служби забезпечують оперативне реагування на події, які порушують функціонування житлової, комунальної та технічної інфраструктури. До таких подій належать пошкодження мереж водо-, тепло-, газо- та електропостачання, затоплення приміщень, несправності ліфтового обладнання, пошкодження будівельних конструкцій та інші ситуації, що потребують втручання відповідних служб. Організаційні засади приймання та передавання повідомлень про екстрені ситуації в Україні визначаються законодавством про функціонування системи допомоги населенню за єдиним номером 112 [1].

Предметна область аварійних служб охоплює процеси прийому повідомлень, реєстрації та класифікації звернень, визначення їх пріоритету, призначення відповідального підрозділу, контролю виконання робіт та інформування заявника. Основною інформаційною одиницею цього процесу є аварійна заявка — формалізований запис про подію, що потребує реагування. Заявка може містити категорію звернення, опис проблеми, адресу або координати місця події, контактні дані заявника, дату створення, пріоритет, поточний статус, відомості про виконавця та додаткові матеріали.

Життєвий цикл заявки починається з надходження повідомлення через телефонний зв’язок, вебформу, електронну пошту, мобільний застосунок або месенджер. Після цього інформація реєструється в системі та перевіряється на повноту. Наступними етапами є класифікація звернення, визначення його пріоритету, призначення відповідальної групи, виконання робіт, перевірка результату та закриття заявки. Подібний порядок відповідає загальній логіці управління інцидентами, яка включає їх реєстрацію, категоризацію, пріоритезацію, призначення, вирішення та формування звітності [4].

Класифікація дає змогу визначити тип аварії та службу, до компетенції якої належить її усунення. Пріоритет заявки доцільно встановлювати з урахуванням терміновості та можливого впливу події на населення або інфраструктуру. Наприклад, витік газу, значний прорив водопроводу або повне припинення електропостачання потребують швидшого реагування, ніж планові ремонтні роботи. У системах управління інцидентами пріоритет зазвичай визначається на основі поєднання впливу та терміновості події [4].

Після класифікації заявка передається відповідальному підрозділу або аварійній бригаді. Маршрутизація може виконуватися оператором або автоматично за категорією звернення, територією обслуговування, завантаженням виконавців та іншими правилами. Автоматизація цього етапу скорочує час між реєстрацією повідомлення та початком робіт, а також зменшує навантаження на диспетчера.

Під час виконання заявка послідовно змінює статуси, наприклад «нова», «прийнята», «призначена», «у роботі», «виконана» та «закрита». За необхідності можуть використовуватися проміжні статуси: «потребує уточнення», «очікує матеріалів» або «передана іншій службі». До заявки додаються службові коментарі, результати огляду, відомості про виконані роботи та фотофіксація. Збереження історії змін дає змогу контролювати дії працівників і відновлювати перебіг виконання звернення.

Основними учасниками процесу є заявник, оператор, виконавець та адміністратор. Заявник повідомляє про подію та отримує інформацію про її опрацювання. Оператор перевіряє звернення, визначає його категорію і передає відповідній службі. Виконавець проводить необхідні роботи та фіксує їх результат. Адміністратор керує обліковими записами, ролями, довідниками та налаштуваннями системи. Керівники підрозділів також можуть використовувати накопичені дані для контролю термінів реагування та оцінювання ефективності роботи.

Однією з основних проблем традиційного прийому заявок є перевантаження телефонних ліній та операторів. Під час усного спілкування

частина відомостей може бути зафіксована неточно, а заявник не завжди має можливість додати фото чи точну геолокацію. Використання месенджера дозволяє організувати послідовний збір інформації, під час якого користувач обирає категорію проблеми, зазначає адресу, додає опис, фотографію або координати. Telegram Bot API, наприклад, підтримує обробку текстових повідомлень, фотографій, файлів і геолокаційних даних [2].

Іншою проблемою є дублювання звернень. Про одну аварію можуть одночасно повідомити декілька користувачів, унаслідок чого оператори створюють кілька заявок на одну подію. Автоматизована система повинна виявляти можливі дублікати за категорією, адресою, часом і змістом повідомлення. Такі заявки можуть об’єднуватися або пов’язуватися з одним аварійним інцидентом, що зменшує кількість повторних виїздів і спрощує диспетчеризацію.

Важливою вимогою є забезпечення зворотного зв’язку. Після телефонного звернення заявник часто не має інформації про реєстрацію та перебіг виконання заявки. Інтеграція з месенджером дозволяє автоматично надсилати номер звернення, повідомляти про його прийняття, призначення виконавця, зміну статусу та завершення робіт. Це підвищує прозорість процесу та скорочує кількість повторних звернень до оператора.

Накопичені дані можуть використовуватися для аналітики. На їх основі визначаються поширені категорії аварій, середній час реагування, кількість заявок за територіями, завантаження підрозділів і дотримання встановлених строків. Отримані результати можуть застосовуватися для планування профілактичних робіт, розподілу ресурсів і виявлення проблемних об’єктів інфраструктури.

Оскільки аварійні звернення можуть мати критичний характер, до інформаційної системи висуваються вимоги щодо надійності, захисту даних, журналювання подій і масштабованості. Система повинна зберігати працездатність у разі зростання кількості звернень та забезпечувати можливість відновлення після збоїв. Для критичних програмних рішень

надійність передбачає застосування моніторингу, резервування, механізмів відновлення та проєктування з урахуванням можливих відмов [5].

Таким чином, діяльність аварійних служб включає взаємопов’язані процеси реєстрації, класифікації, маршрутизації, виконання та контролю заявок. Основними недоліками традиційного підходу є перевантаження операторів, неточність інформації, дублювання звернень і недостатній зворотний зв’язок. Автоматизована система з підтримкою месенджера повинна забезпечувати структуроване збереження даних, контроль життєвого циклу заявки, оперативне інформування учасників і можливість аналізу результатів роботи.

1.2 Огляд існуючих інформаційних систем для обробки звернень і заявок

Для автоматизації прийому, реєстрації та супроводу звернень застосовуються helpdesk-, ticket-, ITSM-, CRM- та муніципальні інформаційні системи. Їх основною інформаційною одиницею є заявка, що містить опис проблеми, дані заявника, категорію, пріоритет, статус, відповідального виконавця та історію виконання. Такі системи забезпечують централізоване зберігання звернень, їх розподіл між виконавцями, контроль строків і формування звітності.

Більшість систем обробки заявок має багаторівневу архітектуру. На рівні взаємодії розміщуються телефонія, електронна пошта, вебпортал, мобільний застосунок і месенджери. Серверна частина реалізує створення, класифікацію, маршрутизацію та зміну статусів заявок. Окремі рівні забезпечують зберігання даних, надсилання сповіщень та формування аналітичної інформації. Узагальнену архітектуру такої системи наведено на рис. 1.1.

Поширеним класом рішень є helpdesk- і ticket-системи. До них належать Zendesk, Freshdesk та інші платформи, які перетворюють звернення з різних каналів на структуровані заявки. Zendesk підтримує централізовану роботу із запитами, що надходять через електронну пошту, чат, соціальні мережі та інші

канали [6]. Система також забезпечує автоматизовану маршрутизацію звернень з урахуванням пріоритету, доступності та завантаження працівників.

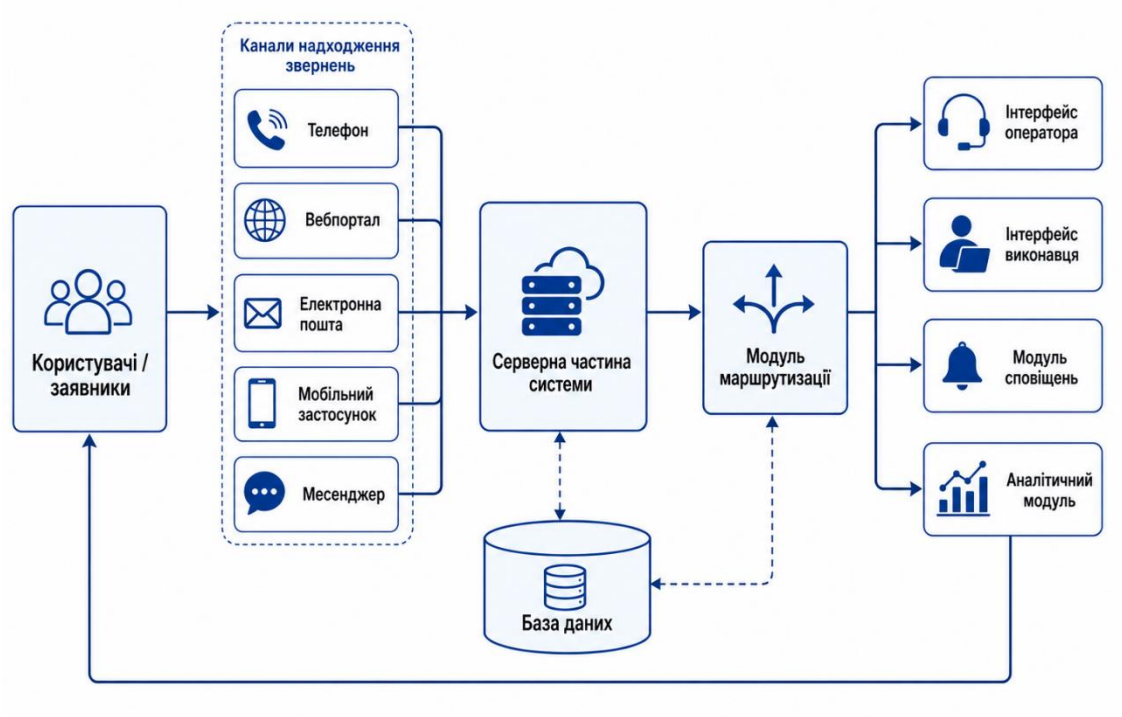

Рисунок 1.1 – Узагальнена архітектура інформаційної системи обробки заявок

Основними перевагами helpdesk-рішень є багатоканальність, єдиний інтерфейс оператора, автоматизація типових дій, збереження історії комунікації та формування звітів. Недоліком є переважна орієнтація на клієнтську підтримку. Для використання в аварійних службах необхідно додатково налаштовувати категорії аварій, територіальну прив’язку, пріоритети, ролі виконавців і спеціальні статуси заявок.

Окрему групу становлять ITSM-системи, зокрема ServiceNow ITSM та Jira Service Management. Вони підтримують управління інцидентами, сервісними запитами, проблемами, змінами та рівнями обслуговування. Типовий процес охоплює реєстрацію інциденту, його класифікацію, пріоритезацію, призначення відповідальній групі, ескалацію, вирішення та

формування звітності [7]. Послідовність обробки звернення в ITSM-системі наведено на рис. 1.2.

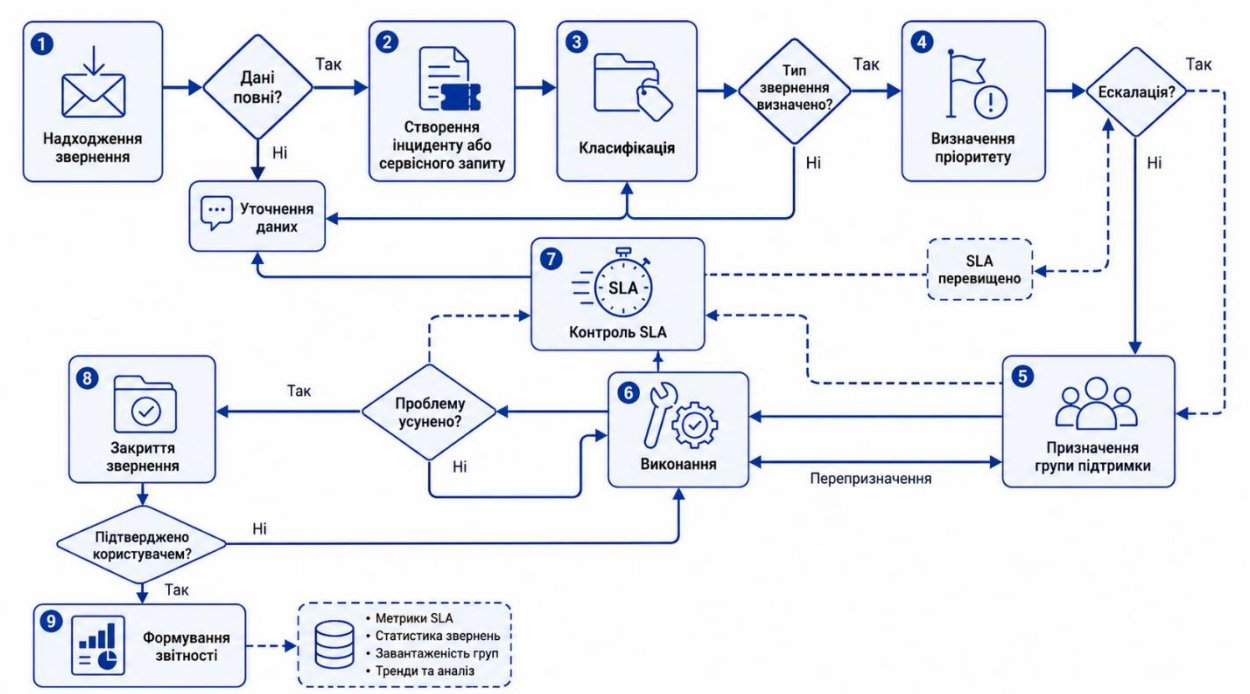

Рисунок 1.2 – Типова схема обробки звернення в ITSM-системі

Jira Service Management організовує вхідні звернення у вигляді черг, які дозволяють операторам переглядати, сортувати, призначати та опрацьовувати заявки [8]. Перевагами платформи є гнучке налаштування типів звернень, статусів, переходів, правил автоматизації та показників SLA. Водночас її використання потребує попереднього моделювання робочих процесів і налаштування під конкретну організацію.

ITSM-рішення забезпечують формалізований контроль строків, ескалацію прострочених заявок та розподіл відповідальності. Це наближує їх до задач аварійних служб. Однак такі платформи переважно орієнтовані на IT-процеси, можуть мати складну конфігурацію та потребувати значних витрат на впровадження, ліцензування і навчання персоналу.

Омніканальні системи об’єднують звернення з телефону, електронної пошти, вебформ, чатів, соціальних мереж і месенджерів у єдиному середовищі. Після надходження звернення створюється заявка, яка

направляється до відповідної черги та призначається виконавцю. Архітектуру омніканальної системи наведено на рис. 1.3. Її перевагою є збереження єдиної історії взаємодії незалежно від використаного каналу, а недоліком — складність інтеграції та підтримки декількох зовнішніх платформ.

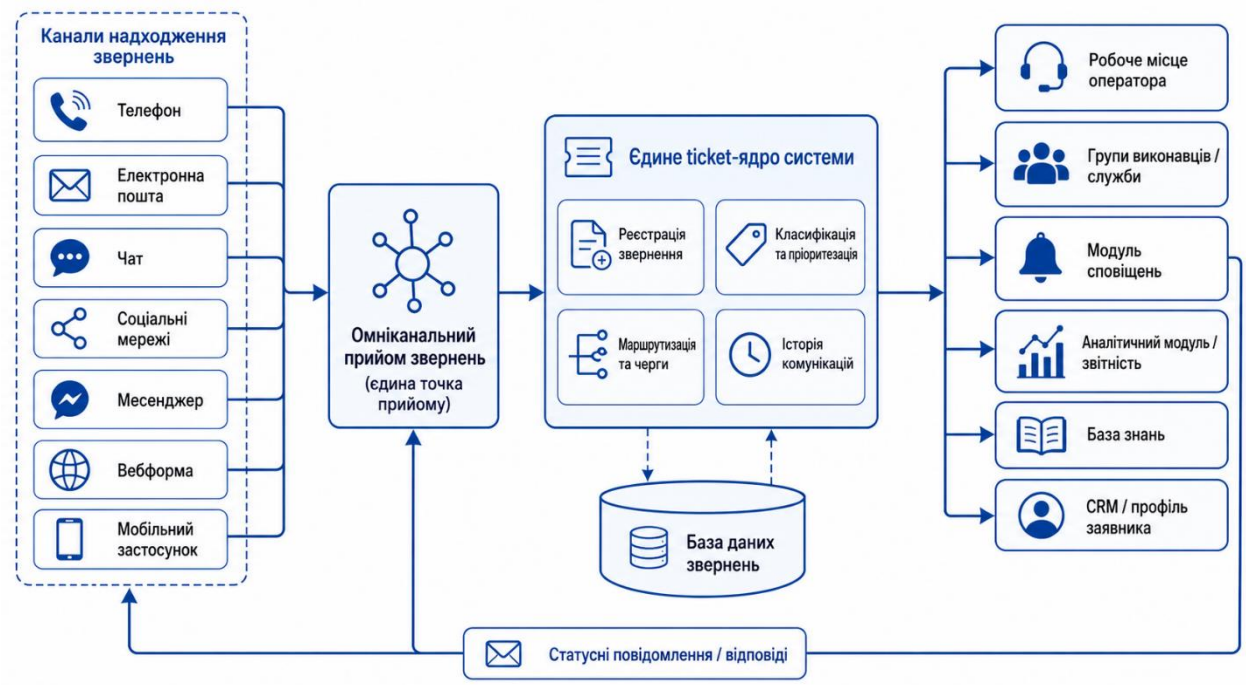

Рисунок 1.3 – Архітектура омніканальної системи обробки звернень

Для роботи зі зверненнями також можуть застосовуватися CRM-системи, зокрема Microsoft Dynamics 365 Customer Service. Вони поєднують облік користувачів, історію взаємодії, кейси, черги, маршрутизацію та аналітику. У Dynamics 365 черги використовуються для групування і розподілу звернень за типом послуги, пріоритетом, категорією або географічною ознакою [9]. Перевагою CRM-підходу є повна історія роботи із заявником, однак значна частина функцій такої платформи не пов’язана безпосередньо з аварійним реагуванням.

Найближчими до предметної області є муніципальні системи звернень громадян. Вони можуть містити вебпортал, мобільний застосунок, карту звернень, адміністративну панель, модуль маршрутизації та засоби контролю виконання. Такі системи враховують територіальну прив’язку подій, проте

часто залежать від конкретного постачальника й не завжди підтримують інтеграцію з месенджерами або незалежне масштабування компонентів.

Порівняння основних типів інформаційних систем наведено в табл. 1.1.

Таблиця 1.1 – Порівняльна характеристика інформаційних систем для обробки звернень і заявок

Тип системи	Основне призначення	Переваги	Недоліки
Helpdesk / ticket-система	Обробка звернень користувачів і клієнтів	Багатоканальність, єдиний інтерфейс оператора, автоматизація, звітність	Орієнтація переважно на клієнтську підтримку, потреба в адаптації до аварійних процесів
ITSM-система	Управління інцидентами, сервісними запитами, SLA	Формалізовані процеси, SLA-контроль, ескалація, аналітика	Висока складність, вартість, орієнтація на IT-сервіси
CRM із модулем кейсів	Облік клієнтів і звернень	Повна історія взаємодії, маршрутизація, інтеграція з контакт-центром	Надмірність функцій, складність налаштування
Муніципальна система звернень	Обробка звернень громадян і комунальних заявок	Прив’язка до міської інфраструктури, карта звернень, публічний контроль	Обмежена гнучкість, залежність від конкретної реалізації
Спеціалізована система аварійних заявок	Оперативне реагування на аварійні ситуації	Орієнтація на аварійні служби, диспетчеризація, контроль виконавців	Часто потребує індивідуальної розробки

Проведений аналіз показує, що існуючі рішення мають спільні функції: реєстрацію заявки, визначення категорії та пріоритету, призначення виконавця, контроль статусу, сповіщення й звітність. Відмінності полягають у сфері застосування, складності впровадження, доступних каналах зв’язку та можливостях адаптації.

1.3 Аналіз використання месенджерів як каналу взаємодії із заявниками

Месенджери є зручним каналом взаємодії між заявниками та аварійними службами, оскільки підтримують передавання текстових повідомлень, фотографій, файлів і геолокації. На відміну від телефонного зв’язку, який потребує постійної участі оператора, месенджер дозволяє автоматизувати первинний збір даних і зменшити навантаження на диспетчерську службу.

У системі обробки аварійних заявок месенджер може використовуватися для створення звернення, уточнення інформації та отримання повідомлень про стан виконання. Користувач через бот обирає категорію аварії, вказує адресу, описує проблему, додає фотографію або геолокацію. Після підтвердження система реєструє звернення, присвоює йому номер і надсилає заявнику відповідне повідомлення. Telegram Bot API підтримує роботу з текстовими повідомленнями, командами, кнопками, файлами, фотографіями та координатами [2].

Взаємодія із системою реалізується через окремий бот-сервіс. Він приймає події від месенджера, керує етапами діалогу, перевіряє введені дані та передає сформовану заявку до серверної частини. Основна бізнес-логіка, маршрутизація і зберігання даних виконуються внутрішніми сервісами системи. Узагальнену схему такої взаємодії наведено на рис. 1.4.

Діалоговий сценарій дозволяє послідовно отримати обов’язкові відомості й зменшити кількість неповних заявок. Типова послідовність включає початок діалогу, вибір типу аварії, введення адреси, опис проблеми, додавання фото або геолокації, підтвердження та створення заявки.

Відповідний сценарій наведено на рис. 1.5, а основні функції бота систематизовано в табл. 1.2.

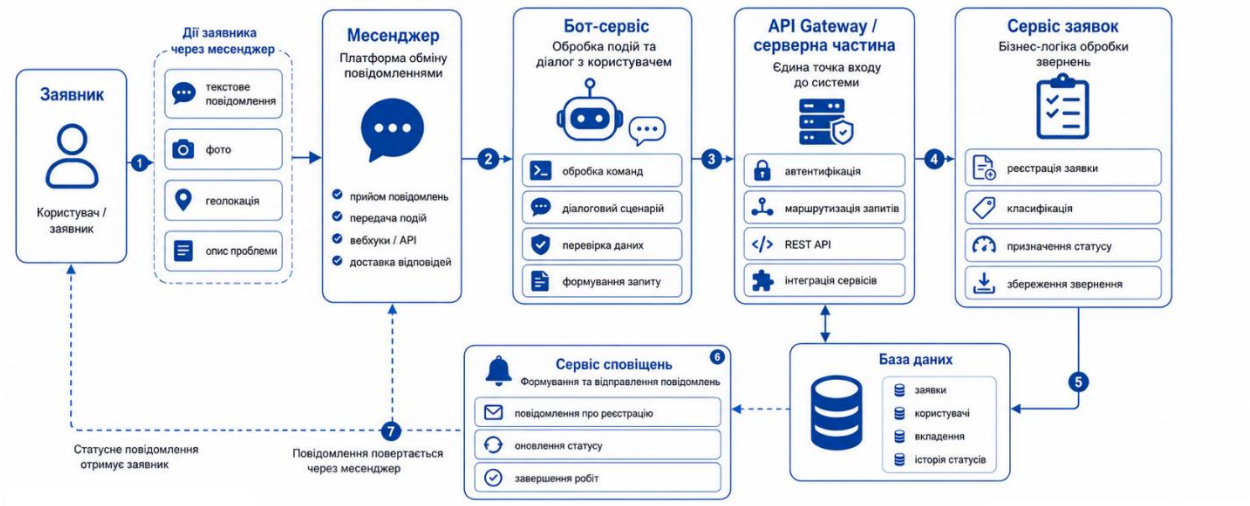

Рисунок 1.4 – Узагальнена схема взаємодії заявника із системою через месенджер

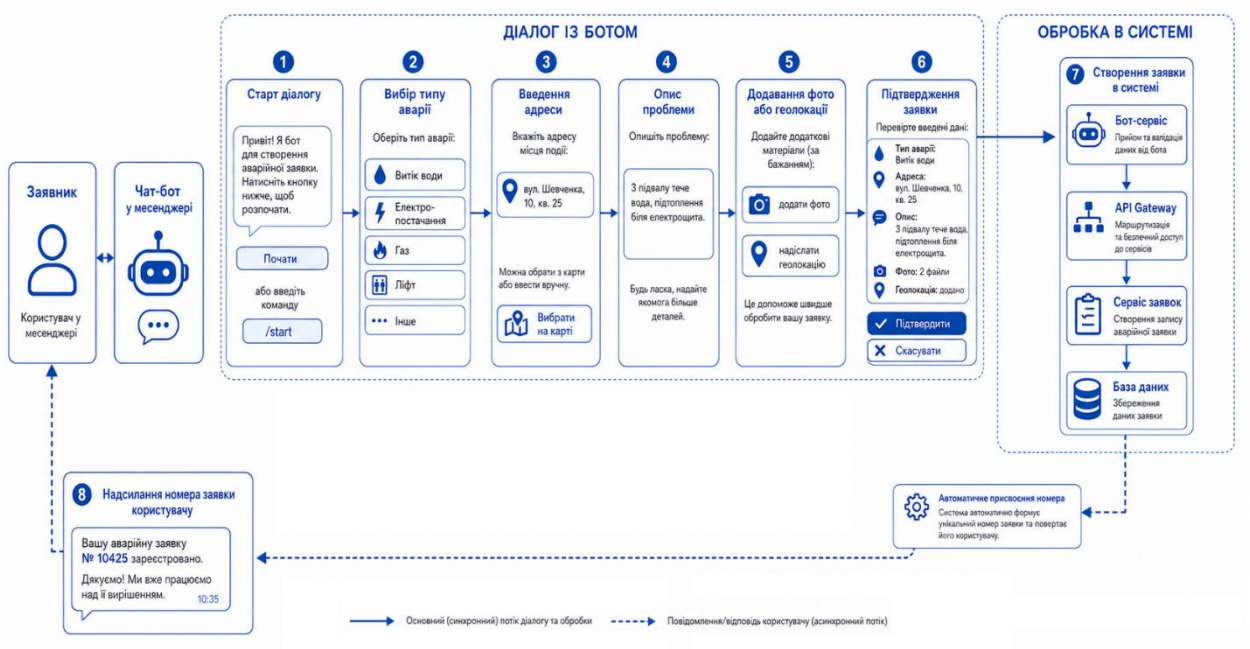

Рисунок 1.5 – Сценарій створення аварійної заявки через месенджер-бот

Перевагою месенджерів є можливість передавання мультимедійних матеріалів. Фотографія пошкодження або координати події допомагають оператору точніше оцінити ситуацію та вибрати відповідну службу. Система

також може автоматично повідомляти користувача про реєстрацію звернення, призначення виконавця, зміну статусу та завершення робіт. Двонаправлений інформаційний обмін між заявником і системою показано на рис. 1.6.

Таблиця 1.2 – Основні функції месенджер-бота в системі обробки аварійних заявок

Функція бота	Призначення	Очікуваний результат
Початок роботи із заявником	Ініціалізація діалогу, виведення головного меню	Користувач отримує доступ до основних дій
Створення нової заявки	Послідовний збір даних про аварійну ситуацію	У системі формується нова заявка
Вибір категорії проблеми	Класифікація звернення за типом аварії	Заявка отримує відповідну категорію
Отримання адреси або геолокації	Визначення місця виникнення проблеми	Заявка прив’язується до конкретного об’єкта або території
Прийом фото або файлів	Отримання додаткових матеріалів для оцінки ситуації	До заявки додаються вкладення
Перевірка статусу заявки	Надання користувачу інформації про хід виконання	Заявник отримує актуальний стан звернення
Надсилання сповіщень	Інформування про зміну статусу або завершення робіт	Забезпечується зворотний зв’язок
Передача звернення оператору	Перехід до ручної обробки у складних випадках	Заявка уточнюється або обробляється працівником служби

Разом із перевагами використання месенджера має певні обмеження. Система залежить від доступності зовнішньої платформи та її API, тому необхідно передбачити журналювання помилок, повторну доставку повідомлень і резервні канали зв’язку. Також слід забезпечити захист контактних даних, адрес, фотографій та інших відомостей, що надходять від

користувача. Доступ до заявок повинен надаватися відповідно до ролей працівників.

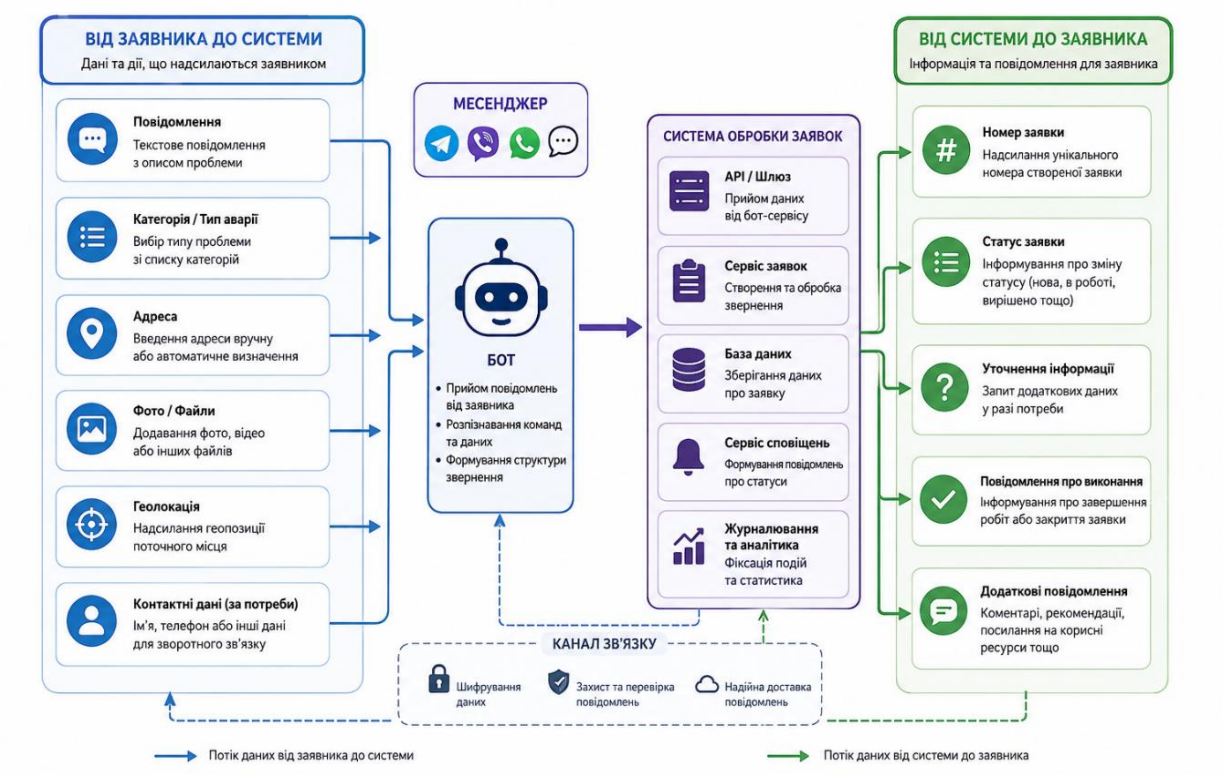

Рисунок 1.6 – Інформаційний обмін між заявником і системою через месенджер

Окремою проблемою є ідентифікація заявника. Ідентифікатора облікового запису месенджера може бути недостатньо, тому система за потреби повинна запитувати номер телефону або інші дані. При цьому кількість обов'язкових полів має бути мінімальною, щоб не ускладнювати створення заявки. Порівняння месенджера з телефоном, електронною поштою та вебформою наведено в табл. 1.3.

У мікросервісній архітектурі бот-сервіс доцільно реалізувати як адаптер між платформою месенджера та внутрішніми сервісами. Сервіс заявок повинен отримувати уніфіковані дані через API і не залежати від конкретного каналу. Це дозволить надалі підключати вебпортал, мобільний застосунок або інший месенджер без зміни основної бізнес-логіки. Поділ системи на

незалежні сервіси забезпечує їх окреме розгортання та масштабування [3]. Місце бот-сервісу в архітектурі показано на рис. 1.7.

Таблиця 1.3 – Порівняння каналів подання аварійних заявок

Критерій порівняння	Телефон	Електронна пошта	Вебформа	Месенджер
Швидкість подання заявки	Висока, але залежить від доступності оператора	Середня	Середня	Висока
Потреба в операторі на першому етапі	Висока	Низька	Низька	Низька або відсутня
Структурованість даних	Низька	Низька або середня	Висока	Висока за умови сценарію бота
Можливість додати фото	Обмежена	Так	Так	Так
Можливість надіслати геолокацію	Обмежена	Обмежена	Можлива	Так
Автоматичне інформування про статус	Потребує додаткового дзвінка або SMS	Можливе	Можливе	Так
Зручність для користувача	Залежить від доступності лінії	Середня	Середня	Висока
Ризик перевантаження каналу	Високий під час масових звернень	Середній	Середній	Нижчий за умови автоматизації
Придатність для автоматизації	Обмежена	Середня	Висока	Висока

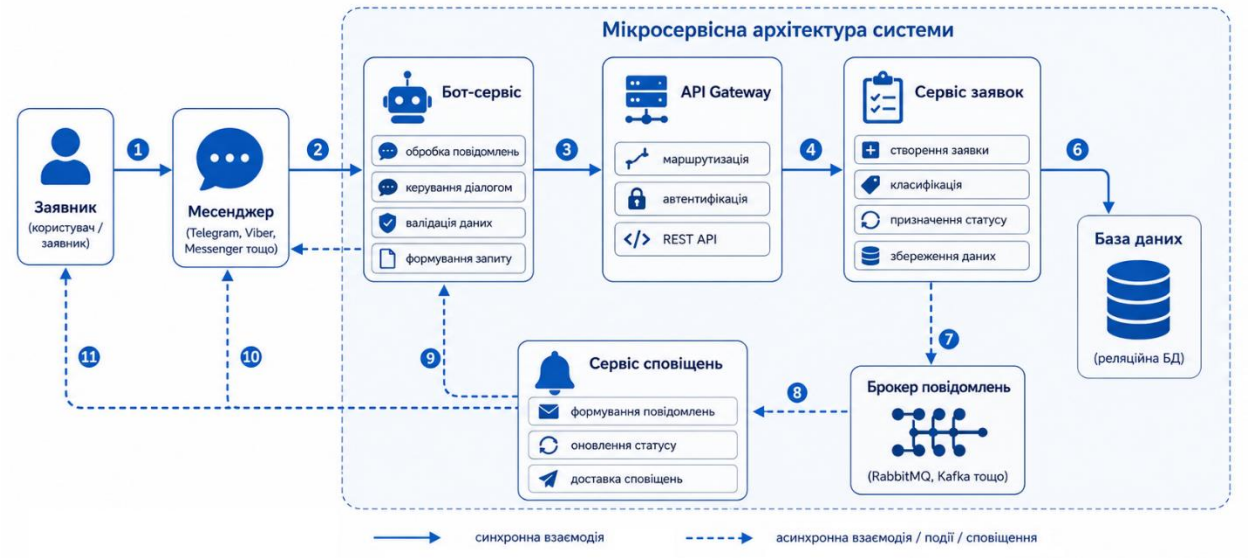

Рисунок 1.7 – Місце месенджер-бота в мікросервісній архітектурі системи обробки заявок

Отже, використання месенджера дозволяє автоматизувати збір інформації, приймати мультимедійні дані, зменшити навантаження на операторів і забезпечити оперативний зворотний зв’язок. Водночас необхідно враховувати безпеку даних, надійність зовнішньої платформи та можливість ручного опрацювання нестандартних заявок. Тому месенджер доцільно використовувати як один із каналів омніканальної системи обробки звернень.

1.4 Аналіз архітектурних підходів до побудови систем обробки заявок

Архітектура інформаційної системи визначає структуру програмних компонентів, способи їх взаємодії, можливості масштабування, надійність і зручність подальшого супроводу. Для системи обробки аварійних заявок вона повинна забезпечувати прийом звернень із різних каналів, збереження та класифікацію заявок, призначення виконавців, зміну статусів, надсилання сповіщень і формування аналітичної інформації.

Найпростішим підходом є монолітна архітектура, у якій інтерфейс, бізнес-логіка, робота з даними, сповіщення та звітність реалізуються в межах одного застосунку. Її перевагами є відносна простота розробки, тестування та

розгортання. Такий підхід доцільний для невеликих систем із обмеженим набором функцій. Однак зі зростанням системи моноліт ускладнює незалежне масштабування модулів, а внесення змін до одного компонента може впливати на роботу всього застосунку. Узагальнену структуру монолітної системи наведено на рис. 1.8.

Рисунок 1.8 – Монолітна архітектура системи обробки заявок

Клієнт-серверна архітектура передбачає поділ системи на клієнтські застосунки та серверну частину. Клієнтами можуть бути вебінтерфейс заявника, панель оператора, мобільний застосунок виконавця або месенджер-бот. Серверна частина реалізує основну бізнес-логіку та взаємодіє з базою даних. Перевагою такого підходу є підтримка декількох типів клієнтів і централізоване керування даними. Водночас сервер може поступово перетворитися на великий моноліт, який складно масштабувати та оновлювати. Схему клієнт-серверної архітектури представлено на рис. 1.9.

Сервісно-орієнтована архітектура передбачає поділ системи на окремі сервіси, які взаємодіють через стандартизовані інтерфейси або корпоративну шину даних. У системі обробки заявок можна виділити сервіси користувачів, заявок, сповіщень, інтеграції та аналітики. Такий підхід полегшує підключення зовнішніх систем і повторне використання функцій. Його недоліками є складніше проєктування, необхідність визначення меж сервісів і

налаштування механізмів обміну даними, безпеки та моніторингу. Сервісно-орієнтовану структуру наведено на рис. 1.10.

Рисунок 1.9 – Клієнт-серверна архітектура системи обробки заявок

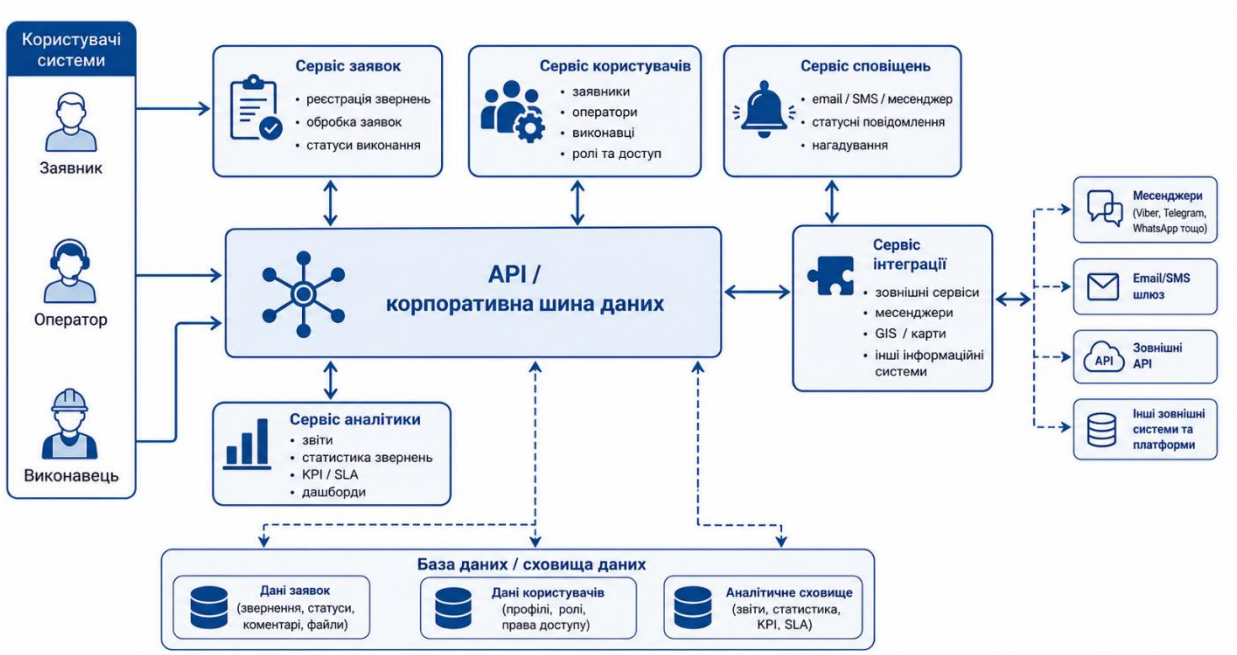

Рисунок 1.10 – Сервісно-орієнтована архітектура системи обробки заявок

Мікросервісна архітектура є розвитком сервісного підходу. Система поділяється на невеликі автономні компоненти, кожен із яких відповідає за окрему функціональну область і може незалежно розроблятися, розгортатися та масштабуватися [3]. Для проєктованої системи доцільно виділити бот-

сервіс, сервіс заявок, сервіс користувачів, сервіс маршрутизації, сервіс сповіщень та аналітичний модуль.

Основною перевагою мікросервісного підходу є можливість окремого масштабування найбільш навантажених компонентів. Наприклад, під час значного збільшення кількості звернень можна масштабувати бот-сервіс і сервіс заявок без зміни аналітичного модуля. Відмова одного компонента також не обов’язково призводить до повної зупинки системи. Якщо сервіс сповіщень тимчасово недоступний, повідомлення можуть бути збережені в черзі та оброблені після відновлення його роботи.

Окремий бот-сервіс виконує роль адаптера між месенджером і внутрішніми компонентами системи. Завдяки цьому сервіс заявок не залежить від конкретної платформи, а підключення нового месенджера, вебпорталу або мобільного застосунку не потребує зміни основної бізнес-логіки.

Недоліком мікросервісної архітектури є вища складність реалізації. Необхідно організувати взаємодію між сервісами, автентифікацію, журналювання, моніторинг, обробку помилок і розподілене зберігання даних. Тому її використання виправдане у випадках, коли система має розвиватися, підтримувати декілька каналів взаємодії та працювати в умовах змінного навантаження. Функціональну схему обраної архітектури наведено на рис. 1.11.

Порівняння розглянутих підходів наведено в табл. 1.4. Монолітна архітектура є найпростішою, але має обмежені можливості масштабування. Клієнт-серверний підхід підтримує різні інтерфейси, проте зберігає залежність від єдиної серверної частини. Сервісно-орієнтована архітектура полегшує інтеграцію, але може містити великі взаємозалежні сервіси. Мікросервісна архітектура є складнішою в реалізації, однак забезпечує найбільшу модульність, гнучкість та незалежність компонентів.

Рисунок 1.11 – Мікросервісна архітектура системи обробки аварійних заявок

Таблиця 1.4 – Порівняння архітектурних підходів до побудови систем обробки заявок

Архітектурний підхід	Сутність підходу	Переваги	Недоліки	Доцільність для системи аварійних заявок
1	2	3	4	5
Монолітна архітектура	Усі функції реалізуються в межах одного застосунку	Простота розробки, тестування та розгортання; зручність для невеликих систем	Складність масштабування окремих модулів; сильна зв’язаність компонентів; ризик впливу змін на всю систему	Доцільна лише для простого прототипу або невеликої локальної системи

Продовження табл. 1.4

1	2	3	4	5
Клієнт-серверна архітектура	Клієнтські застосунки взаємодіють із серверною частиною через API	Розділення інтерфейсу та бізнес-логіки; можливість підтримки кількох клієнтів	Серверна частина може стати великим монолітом; обмежене незалежне масштабування	Придатна для систем середньої складності, але має обмеження для подальшого масштабування
Сервісно-орієнтована архітектура	Система складається з окремих сервісів, які взаємодіють через стандартизовані інтерфейси	Модульність, інтеграція з іншими системами, повторне використання сервісів	Складніше проєктування та адміністрування; можливість надмірної зв’язаності сервісів	Доцільна для інтеграційних систем і великих організацій
Мікросервісна архітектура	Система поділяється на автономні сервіси з чіткими межами відповідальності	Незалежне масштабування, гнучкість розвитку, відмовостійкість, зручність інтеграції з месенджерами	Вища складність розгортання, моніторингу, обміну даними та тестування	Найбільш доцільна для гнучкої системи аварійних заявок з кількома каналами взаємодії

Отже, для системи обробки заявок аварійних служб обрано мікросервісну архітектуру. Вона відповідає функціональному поділу системи, підтримує інтеграцію з месенджерами, дозволяє незалежно масштабувати окремі сервіси та забезпечує основу для подальшого розширення програмного рішення.

Висновки до розділу:

У першому розділі проаналізовано предметну область аварійних служб, існуючі системи обробки звернень, можливості використання месенджерів і основні архітектурні підходи. Встановлено, що заявка є центральною інформаційною одиницею та проходить етапи реєстрації, класифікації, маршрутизації, виконання і закриття.

Основними недоліками традиційної обробки звернень є перевантаження операторів, неточність фіксації даних, дублювання заявок, недостатній зворотний зв’язок і обмежені можливості аналітики. Існуючі helpdesk-, ITSM-, CRM- та муніципальні системи частково вирішують ці проблеми, однак потребують адаптації до специфіки аварійних служб.

Використання месенджера дозволяє автоматизувати первинний збір даних, приймати фото та геолокацію, а також інформувати заявника про стан звернення. Найбільш доцільною архітектурою визначено мікросервісну, оскільки вона забезпечує модульність, незалежне масштабування компонентів і зручну інтеграцію із зовнішніми каналами.

На основі проведеного аналізу сформульовано задачу розробки системи, що забезпечує прийом заявок через месенджер, їх збереження, класифікацію, визначення пріоритету, маршрутизацію до відповідальної служби, зміну статусів і надсилання повідомлень користувачам.

До основних функціональних вимог належать створення заявки, введення адреси й опису проблеми, додавання фото або геолокації, перегляд і зміна статусу, призначення виконавця, ведення історії дій та формування звітності. Нефункціональними вимогами є надійність, масштабованість, захист даних, розмежування доступу, журналювання подій і можливість подальшого розширення системи.

Отже, розробка спеціалізованої системи на основі мікросервісної архітектури та месенджера є доцільною для підвищення оперативності й прозорості обробки аварійних заявок.

РОЗДІЛ 2
ПРОЄКТУВАННЯ СИСТЕМИ ОБРОБКИ ЗАЯВОК ДЛЯ АВАРІЙНИХ СЛУЖБ НА ОСНОВІ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

2.1 Обґрунтування вибору технологій реалізації системи

Реалізація системи обробки заявок для аварійних служб потребує технологій, які забезпечують модульність, надійність, масштабованість та інтеграцію з месенджером. Оскільки систему вирішено будувати на основі мікросервісної архітектури, окремі компоненти повинні мати чіткі функції, підтримувати незалежне розгортання та взаємодіяти через стандартизовані інтерфейси.

Для розробки серверної частини обрано ASP.NET Core — кросплатформний фреймворк із відкритим кодом, призначений для створення вебзастосунків, REST API та мережевих сервісів. Він підтримує вбудоване впровадження залежностей, конфігурацію, маршрутизацію, автентифікацію, авторизацію та журналювання [10]. У межах проєкту на його основі можуть бути реалізовані сервіси заявок, користувачів, маршрутизації, сповіщень і взаємодії з месенджером.

Для доступу до даних доцільно застосувати Entity Framework Core. Цей об’єктно-реляційний засіб дозволяє працювати з базою даних через класи та об’єкти .NET, зменшуючи обсяг ручного SQL-коду. Підтримка міграцій спрощує створення та подальшу зміну структури бази даних [11].

Як систему керування базами даних обрано PostgreSQL. Вона підтримує транзакції, індекси, обмеження цілісності та складні SQL-запити [12]. Реляційна модель відповідає структурі предметної області, у якій користувачі, заявки, статуси, типи аварій, служби, виконавці та повідомлення мають чіткі зв’язки. Для проєкту доцільно використати одну СКБД із логічним поділом таблиць за сервісами, що спрощує розгортання та тестування.

Синхронну взаємодію між клієнтами й сервісами доцільно реалізувати через REST API. Інтерфейс оператора може використовувати його для

перегляду заявок, призначення виконавців і зміни статусів. Зовнішні запити надходитимуть через API Gateway, який виконує роль єдиної точки входу та маршрутизує їх до відповідних сервісів. Для його реалізації може бути використано YARP — гнучку бібліотеку зворотного проксі для платформи .NET [13].

Для подій, які не потребують негайної відповіді, доцільно застосувати асинхронну взаємодію через RabbitMQ. Брокер повідомлень дозволяє передавати події між сервісами через черги та зменшує їх взаємну залежність [14]. Наприклад, після створення заявки сервіс заявок формує подію RequestCreated, яку отримують сервіси маршрутизації та сповіщень. Зміна статусу може створювати подію RequestStatusChanged, на основі якої користувачу надсилається повідомлення.

Для прийому заявок через месенджер обрано Telegram Bot API. Він є HTTP-інтерфейсом для створення ботів і підтримує текстові повідомлення, команди, кнопки, фотографії, файли та геолокацію [2]. Бот забезпечуватиме послідовний збір даних про тип аварії, адресу, опис проблеми й додаткові матеріали. Після реєстрації звернення користувач отримуватиме номер заявки та повідомлення про зміну її стану.

Розгортання компонентів системи доцільно виконувати за допомогою Docker. Кожний мікросервіс, база даних і брокер повідомлень можуть запускатися в окремому контейнері. Для керування багатоконтейнерним застосунком використовується Docker Compose, який дозволяє описати сервіси, мережі, порти та сховища в одному конфігураційному файлі [15]. Такий підхід забезпечує однакове середовище запуску під час розробки, тестування та демонстрації системи.

Для документування REST API доцільно використовувати OpenAPI та Swagger UI. Специфікація OpenAPI описує доступні маршрути, параметри, моделі запитів і відповідей, а також способи автентифікації [16]. Інтерактивний інтерфейс Swagger спрощує перевірку реалізованих методів і представлення результатів у практичній частині роботи.

Інтерфейс оператора може бути реалізований засобами ASP.NET Core MVC або Razor Pages. Він повинен забезпечувати перегляд і фільтрацію заявок, призначення виконавців, зміну статусів та перегляд історії обробки. Для захисту системи необхідно передбачити рольову модель доступу для оператора, виконавця й адміністратора, а також журналювання створення заявок, зміни статусів, помилок і дій користувачів.

Узагальнений вибір технологій наведено в табл. 2.1.

Таблиця 2.1 – Обґрунтування вибору технологій реалізації системи

Компонент системи	Обрана технологія	Призначення в системі	Обґрунтування вибору
1	2	3	4
Серверна частина та мікросервіси	ASP.NET Core	Реалізація REST API, бізнес-логіки та окремих сервісів	Висока продуктивність, підтримка Web API, dependency injection, зручність створення мікросервісів
Доступ до даних	Entity Framework Core	Робота з базою даних через об’єктні моделі	Підтримка міграцій, зменшення обсягу SQL-коду, інтеграція з ASP.NET Core
База даних	PostgreSQL	Збереження заявок, користувачів, статусів, повідомлень і журналів	Надійність, підтримка транзакцій, відкритий код, придатність для реляційних моделей
Асинхронна взаємодія	RabbitMQ	Передача подій між мікросервісами	Підтримка черг повідомлень, зменшення зв’язаності компонентів, стійкість до тимчасових збоїв

Продовження табл 2.1

1	2	3	4
Інтеграція з месенджером	Telegram Bot API	Прийом заявок і надсилання повідомлень через месенджер	Підтримка ботів, кнопок, фото, геолокації, вебхуків і автоматизованих сценаріїв
Єдина точка входу	API Gateway / YARP	Маршрутизація зовнішніх запитів до мікросервісів	Приховування внутрішньої структури системи, централізація доступу, зручність масштабування
Контейнеризація	Docker, Docker Compose	Розгортання сервісів у незалежних контейнерах	Ізоляція компонентів, спрощення запуску, єдине середовище розробки та тестування
Документування API	Swagger / OpenAPI	Опис і тестування REST API	Автоматична документація, зручність перевірки API-методів
Інтерфейс оператора	ASP.NET Core MVC / Razor Pages	Адміністративна панель для роботи із заявками	Швидка розробка вебінтерфейсу, інтеграція із серверною логікою
Безпека та доступ	ASP.NET Core Identity / JWT	Автентифікація, авторизація, розмежування ролей	Підтримка рольової моделі доступу, захист API та користувацьких даних
Журналювання	ASP.NET Core Logging / Serilog	Фіксація подій і помилок системи	Контроль роботи сервісів, діагностика помилок, аудит дій користувачів

Обраний стек — ASP.NET Core, Entity Framework Core, PostgreSQL, RabbitMQ, Telegram Bot API, YARP і Docker — відповідає вимогам мікросервісної системи та забезпечує можливість її подальшого розширення.

2.2 Проєктування загальної архітектури системи

Загальна архітектура системи обробки аварійних заявок проєктується на основі мікросервісного підходу. Така архітектура передбачає поділ системи на окремі компоненти, кожен із яких виконує власну функцію та може незалежно розгортатися і масштабуватися [3]. Це є доцільним для аварійних служб, оскільки система повинна підтримувати прийом звернень, їх обробку, маршрутизацію, сповіщення та аналітику.

Основними користувачами системи є заявник, оператор, виконавець та адміністратор. Заявник подає звернення через месенджер, оператор контролює та розподіляє заявки, виконавець отримує призначені завдання і змінює їх статус, а адміністратор керує обліковими записами та довідниками системи.

Єдиною точкою входу до серверної частини є API Gateway, який приймає зовнішні запити та маршрутизує їх до відповідних сервісів [13]. Такий підхід спрощує доступ до системи, централізує логіку маршрутизації та приховує внутрішню структуру мікросервісів. Загальну архітектуру системи доцільно подати на рис. 2.1.

У складі системи доцільно виділити такі основні мікросервіси:

– бот-сервіс, який забезпечує взаємодію з месенджером і приймає звернення користувача;

– сервіс заявок, який створює, зберігає та оновлює заявки;

– сервіс користувачів, який відповідає за облікові записи, ролі та права доступу;

– сервіс маршрутизації, який визначає відповідальну службу або виконавця;

– сервіс сповіщень, який надсилає повідомлення користувачам і працівникам;

– аналітичний сервіс, який формує узагальнену інформацію та звітні дані.

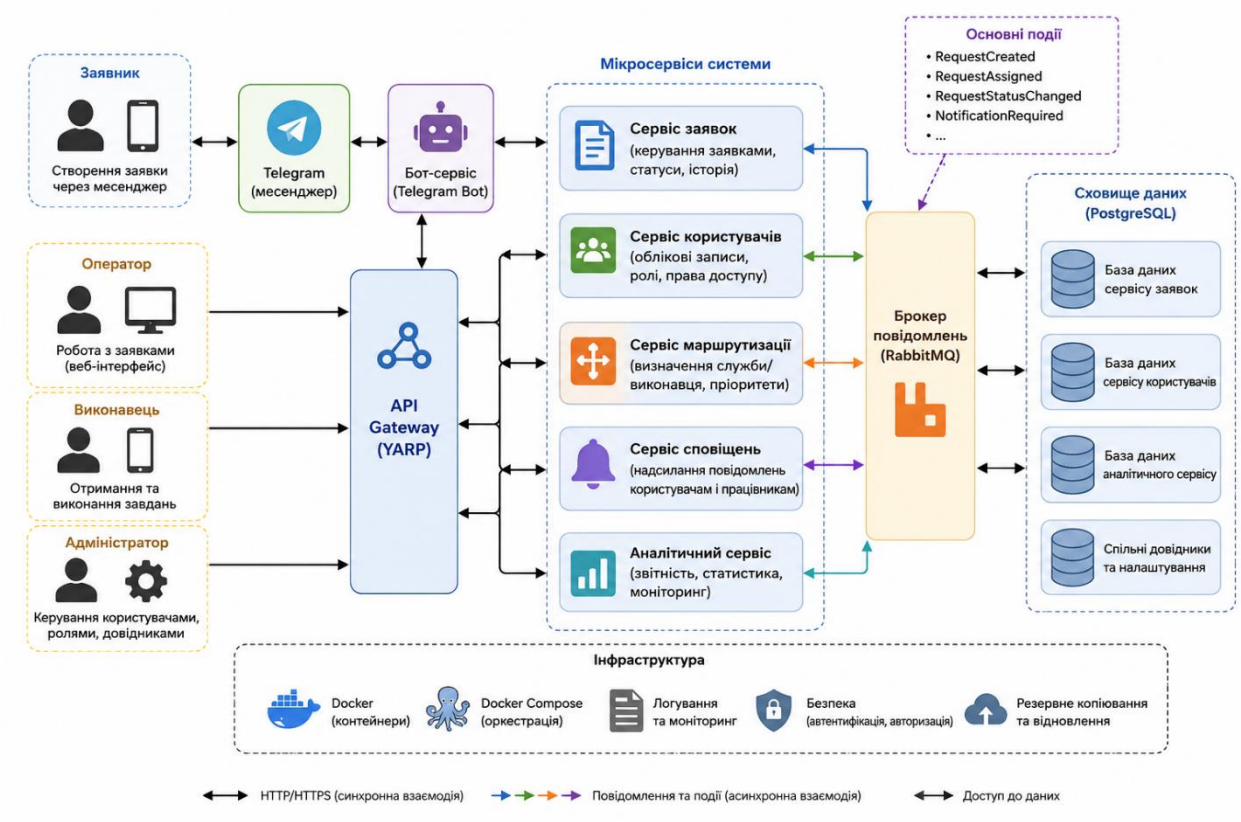

Рисунок 2.1 – Загальна архітектура системи обробки аварійних заявок на основі мікросервісного підходу

Центральним компонентом є сервіс заявок, оскільки він працює з основною сутністю системи — аварійною заявкою. Після надходження звернення через бот-сервіс дані передаються через API Gateway до сервісу заявок, де проходять перевірку та зберігаються в базі даних. Після реєстрації формується подія про створення нової заявки.

Для синхронної взаємодії між клієнтами та сервісами використовується REST API, а для асинхронного обміну подіями — RabbitMQ [14]. Це дозволяє зменшити зв’язаність компонентів. Наприклад, після створення заявки сервіс заявок може опублікувати подію RequestCreated, яку отримують сервіс маршрутизації та сервіс сповіщень. Після зміни статусу формується подія RequestStatusChanged, на основі якої користувач отримує повідомлення.

Для зберігання інформації використовується PostgreSQL. У межах навчального проєкту допускається використання однієї бази даних із логічним

поділом таблиць між сервісами, хоча в повноцінній мікросервісній системі кожен сервіс може мати власне сховище [17].

Оператор працює із системою через вебінтерфейс, який також звертається до серверної частини через API Gateway. Через цей інтерфейс можна переглядати заявки, фільтрувати їх, призначати виконавців, змінювати статуси та переглядати історію виконання.

Узагальнений сценарій роботи системи є таким: заявник створює звернення через месенджер; бот-сервіс передає дані до сервісу заявок; сервіс маршрутизації визначає відповідальну службу; сервіс сповіщень інформує користувача; оператор і виконавець працюють із заявкою до її завершення. Після цього дані можуть використовуватися для аналітики та звітності.

Розгортання системи доцільно виконувати в Docker-контейнерах, а для локального запуску використовувати Docker Compose [15]. Це спрощує тестування та забезпечує однакове середовище виконання для всіх компонентів.

Отже, запропонована архітектура поєднує API Gateway, набір спеціалізованих мікросервісів, брокер повідомлень і базу даних. Такий підхід забезпечує модульність, гнучкість, можливість масштабування та зручність подальшого розвитку системи.

2.3 Проєктування бізнес-процесу обробки заявки

Бізнес-процес обробки аварійної заявки охоплює всі етапи її життєвого циклу: від створення звернення до завершення робіт і закриття заявки. Основною метою процесу є забезпечення повноти вхідних даних, своєчасної маршрутизації звернення, контролю виконання та інформування заявника.

Процес починається зі створення звернення через месенджер. Заявник обирає категорію аварії, зазначає адресу, додає опис проблеми, фотографію або геолокацію. Бот-сервіс перевіряє наявність обов’язкових даних. Якщо інформація є неповною, користувачу надсилається запит на уточнення.

Після підтвердження дані передаються через API Gateway до сервісу заявок. Сервіс створює запис у базі даних, присвоює заявці унікальний номер і встановлює початковий статус «Нова». Одночасно публікується подія RequestCreated, яка використовується іншими компонентами системи.

Наступним етапом є класифікація заявки та визначення її пріоритету. Сервіс маршрутизації встановлює відповідальну службу або виконавця з урахуванням типу аварії, місця події та доступних ресурсів. Якщо автоматичне призначення неможливе, заявка передається оператору для ручного опрацювання. Після призначення формується подія RequestAssigned, а статус заявки змінюється на «Призначена».

Виконавець приймає заявку та переводить її у статус «У роботі». У процесі виконання можуть додаватися службові коментарі, результати огляду та фотофіксація. Якщо необхідні додаткові ресурси або уточнення, заявка може бути перепризначена чи повернена на попередній етап. Контроль строків здійснюється відповідно до пріоритету заявки та встановлених нормативів. У разі перевищення допустимого часу система формує повідомлення про ескалацію [7].

Після завершення робіт виконавець фіксує результат, а заявка отримує статус «Виконана». Сервіс сповіщень надсилає користувачу повідомлення про завершення. Якщо заявник підтверджує усунення проблеми, статус змінюється на «Закрита». У разі негативного результату заявка повертається на етап виконання.

Для заявки передбачено такі основні статуси: «Нова», «Призначена», «У роботі», «Очікує уточнення», «Виконана», «Закрита» та «Скасована». Усі зміни статусів, виконавців і службових даних зберігаються в історії, що забезпечує аудит дій і можливість подальшого формування звітності.

Функціональну схему бізнес-процесу наведено на рис. 2.2. На ній відображено ролі заявника, бот-сервісу, сервісу заявок, сервісу маршрутизації, оператора, виконавця та сервісу сповіщень, а також основні рішення й переходи між етапами.

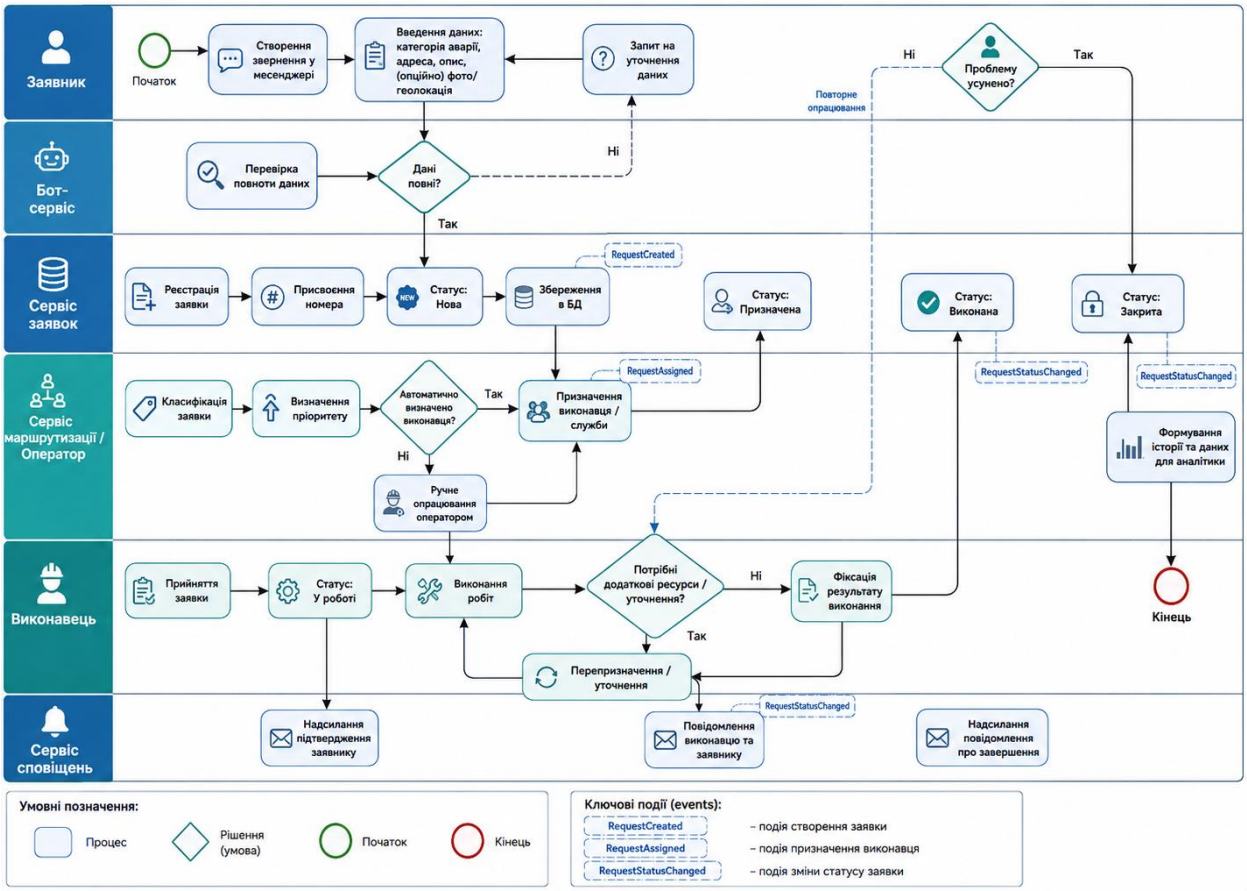

Рисунок 2.2 – Бізнес-процес обробки аварійної заявки в інформаційній системі

Асинхронна взаємодія між компонентами реалізується через події RequestCreated, RequestAssigned і RequestStatusChanged, що передаються за допомогою брокера повідомлень. Це зменшує залежність між сервісами та забезпечує своєчасне інформування користувачів [14].

Отже, спроєктований бізнес-процес охоплює повний цикл обробки аварійної заявки, визначає відповідальність учасників і забезпечує контроль переходів між статусами.

2.4 Проєктування структури бази даних

База даних системи призначена для зберігання інформації про аварійні заявки, користувачів, служби, повідомлення, вкладення та історію виконання. Для її реалізації обрано реляційну СКБД PostgreSQL, яка підтримує транзакції, зовнішні ключі, обмеження цілісності та індексування даних [12].

Проєктування реляційної бази даних передбачає визначення сутностей, їх атрибутів, первинних і зовнішніх ключів та зв'язків між таблицями. Розподіл даних між окремими сутностями дає змогу зменшити дублювання інформації та забезпечити її цілісність [18, 19].

Центральною сутністю є аварійна заявка. Вона пов'язується із заявником, типом аварії, поточним статусом, відповідальною службою та виконавцем. Окремі таблиці використовуються для зберігання історії змін, повідомлень, вкладень, сповіщень і журналу системних подій (див. рис. 2.3).

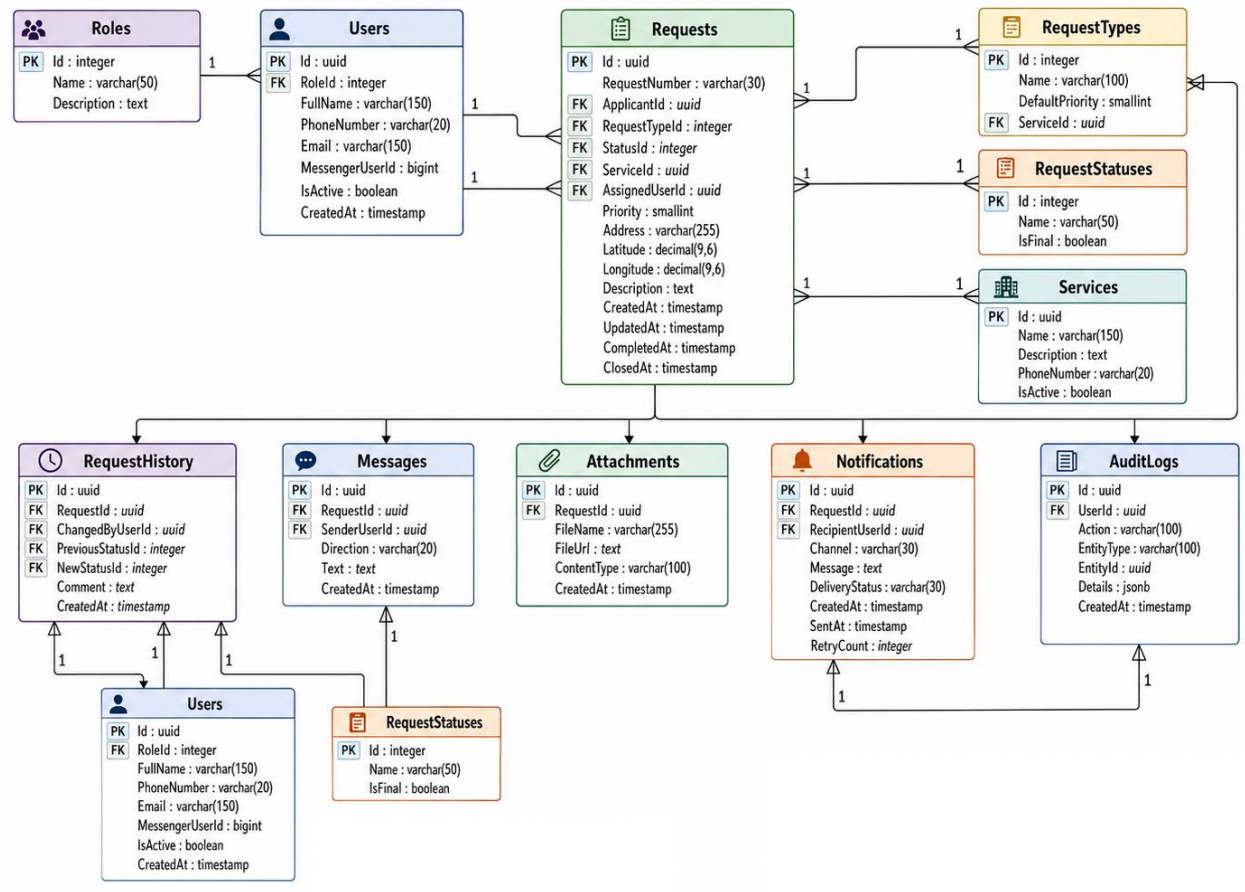

Рисунок 2.3 – ER-модель бази даних системи обробки аварійних заявок

У межах навчального проєкту використовується одна фізична база даних із логічним розподілом таблиць за функціональними областями. Доступ до даних здійснюється через відповідні сервіси. Надалі така структура може бути розділена на окремі сховища мікросервісів [17].

Таблиця Requests (див. табл 2.2) містить основні відомості про аварійні заявки та їх поточний стан. Вона є центральною таблицею бази даних.

Таблиця 2.2 – Структура таблиці Requests

Назва поля	Тип даних	Призначення
Id	uuid	Унікальний ідентифікатор заявки
RequestNumber	varchar(30)	Номер заявки для відображення користувачу
ApplicantId	uuid	Посилання на заявника
RequestTypeId	integer	Тип аварійної ситуації
StatusId	integer	Поточний статус заявки
ServiceId	uuid	Відповідальна аварійна служба
AssignedUserId	uuid	Призначений виконавець
Priority	smallint	Рівень пріоритету
Address	varchar(255)	Адреса місця події
Latitude	decimal(9,6)	Географічна широта
Longitude	decimal(9,6)	Географічна довгота
Description	text	Опис аварійної ситуації
CreatedAt	timestamp	Час реєстрації заявки
UpdatedAt	timestamp	Час останньої зміни
CompletedAt	timestamp	Час завершення робіт
ClosedAt	timestamp	Час закриття заявки

Таблиця Users (див. табл. 2.3) призначена для зберігання даних заявників, операторів, виконавців та адміністраторів. Рівень доступу користувача визначається посиланням на відповідну роль.

Таблиця 2.3 – Структура таблиці Users

Назва поля	Тип даних	Призначення
Id	uuid	Унікальний ідентифікатор користувача
RoleId	integer	Посилання на роль користувача
FullName	varchar(150)	Повне ім’я користувача
PhoneNumber	varchar(20)	Контактний номер телефону
Email	varchar(150)	Адреса електронної пошти
MessengerUserId	bigint	Ідентифікатор у месенджері
PasswordHash	varchar(255)	Хеш пароля внутрішнього користувача
IsActive	boolean	Ознака активності облікового запису
CreatedAt	timestamp	Дата і час створення запису

Поля координат є необов’язковими та заповнюються під час надсилання геолокації через месенджер.

Довідникові таблиці Roles, RequestTypes, RequestStatuses і Services містять довідкові значення, які використовуються під час перевірки доступу, класифікації та маршрутизації заявок.

Таблиця 2.4 – Структура довідникових таблиць

Таблиця	Основні поля	Призначення	Таблиця
Roles	Id, Name, Description	Зберігання ролей користувачів	Roles
RequestTypes	Id, Name, DefaultPriority, ServiceId	Типи аварій та початкові правила маршрутизації	RequestTypes
RequestStatuses	Id, Name, IsFinal	Перелік можливих статусів заявки	RequestStatuses
Services	Id, Name, Description, PhoneNumber, IsActive	Відомості про ава	Services

Таблиця RequestHistory призначена для фіксації змін статусу, виконавця та інших параметрів заявки. Її використання забезпечує відновлення послідовності дій та аудит процесу виконання.

Таблиця Messages (див. табл. 2.6) зберігає історію спілкування із заявником. Таблиця Attachments (див. табл. 2.6) містить метадані фотографій і файлів, пов’язаних із заявкою. Самі файли доцільно зберігати в окремому файловому або об’єктному сховищі.

Таблиця 2.5 – Структура таблиці RequestHistory

Назва поля	Тип даних	Призначення
Id	uuid	Ідентифікатор запису історії
RequestId	uuid	Посилання на заявку
ChangedByUserId	uuid	Користувач, який виконав зміну
PreviousStatusId	integer	Попередній статус
NewStatusId	integer	Новий статус
Comment	text	Коментар до виконаної зміни
CreatedAt	timestamp	Дата і час події

Таблиця 2.6 – Структура таблиць Messages та Attachments

Таблиця	Назва поля	Тип даних	Призначення
Messages	Id	uuid	Ідентифікатор повідомлення
	RequestId	uuid	Посилання на заявку
	SenderUserId	uuid	Відправник повідомлення
	Direction	varchar(20)	Напрям повідомлення
	Text	text	Текст повідомлення
	CreatedAt	timestamp	Час надсилання
Attachments	Id	uuid	Ідентифікатор вкладення
	RequestId	uuid	Посилання на заявку
	FileName	varchar(255)	Початкова назва файлу
	FileUrl	text	Шлях до збереженого файлу
	ContentType	varchar(100)	MIME-тип вкладення
	CreatedAt	timestamp	Час додавання

Таблиця Notifications (див. табл. 2.7) використовується сервісом сповіщень для збереження повідомлень та контролю їх доставки через месенджер або інший канал.

Таблиця 2.7 – Структура таблиці Notifications

Назва поля	Тип даних	Призначення
Id	uuid	Ідентифікатор сповіщення
RequestId	uuid	Посилання на заявку
RecipientUserId	uuid	Одержувач повідомлення
Channel	varchar(30)	Канал надсилання
Message	text	Текст сповіщення
DeliveryStatus	varchar(30)	Стан доставки
CreatedAt	timestamp	Час створення
SentAt	timestamp	Час успішного надсилання
RetryCount	integer	Кількість повторних спроб

Таблиця AuditLogs призначена для журналювання входів до системи, змін заявок, адміністративних операцій і помилок доступу.

Таблиця 2.8 – Структура таблиці AuditLogs

Назва поля	Тип даних	Призначення
Id	uuid	Ідентифікатор події
UserId	uuid	Користувач, пов’язаний із подією
Action	varchar(100)	Назва виконаної операції
EntityType	varchar(100)	Тип зміненої сутності
EntityId	uuid	Ідентифікатор сутності
Details	jsonb	Додаткові відомості про подію
CreatedAt	timestamp	Дата і час події

Для прискорення пошуку доцільно створити індекси за полями RequestNumber, StatusId, ServiceId, AssignedUserId, CreatedAt і MessengerUserId. Первинні та зовнішні ключі забезпечують зв’язки між сутностями, а обмеження унікальності застосовуються до номера заявки й ідентифікатора користувача месенджера.

Отже, спроєктована база даних забезпечує зберігання основних сутностей системи, контроль життєвого циклу заявки, історію комунікації та аудит дій. Її структура відповідає функціональному поділу програмної системи й допускає подальше розділення даних між окремими мікросервісами.

2.5 Проєктування API взаємодії між сервісами

Взаємодія між компонентами системи повинна забезпечувати надійну передачу даних, слабку зв’язаність мікросервісів і можливість їх незалежного розвитку. Для цього в системі поєднуються синхронні REST-запити та асинхронний обмін подіями через брокер повідомлень RabbitMQ.

Зовнішні клієнти, зокрема вебінтерфейс оператора та бот-сервіс, звертаються до системи через API Gateway. Він виконує роль єдиної точки входу, приховує внутрішню структуру мікросервісів і маршрутизує запити до відповідних компонентів [13]. Обмін даними здійснюється у форматі JSON через протокол HTTPS, а структура маршрутів і моделей описується засобами OpenAPI [16].

Основними ресурсами API є заявки, користувачі, служби, вкладення, історія змін і сповіщення. Для роботи з ними використовуються стандартні HTTP-методи: GET — для отримання даних, POST — для створення ресурсів, PATCH — для часткового оновлення та DELETE — для видалення.

Під час створення звернення бот-сервіс формує запит POST /api/v1/requests, що містить тип аварії, адресу, опис, координати, ідентифікатор заявника та відомості про вкладення. Сервіс заявок перевіряє дані, створює запис у базі даних і повертає код 201 Created, унікальний номер та початковий статус заявки.

Таблиця 2.9 – Основні методи REST API системи

HTTP-метод	Маршрут	Призначення
POST	/api/v1/requests	Створення аварійної заявки
GET	/api/v1/requests	Отримання списку заявок
GET	/api/v1/requests/{id}	Отримання даних конкретної заявки
PATCH	/api/v1/requests/{id}/status	Зміна статусу заявки
PATCH	/api/v1/requests/{id}/assign	Призначення служби або виконавця
GET	/api/v1/requests/{id}/history	Отримання історії змін
POST	/api/v1/requests/{id}/attachments	Додавання вкладення
GET	/api/v1/users/{id}	Отримання даних користувача
GET	/api/v1/services	Отримання переліку служб
POST	/api/v1/bot/webhook	Приймання подій від месенджера

Послідовність взаємодії компонентів під час створення аварійної заявки доцільно подати у вигляді UML-діаграми послідовності на рис. 2.4.

Як показано на рис. 2.4, заявник передає повідомлення через Telegram, після чого бот-сервіс структурує отримані дані та надсилає їх через API Gateway до сервісу заявок. Після збереження звернення в базі даних сервіс повертає номер заявки та публікує подію RequestCreated у брокері повідомлень. Сервіс маршрутизації отримує цю подію, визначає відповідальну службу або виконавця та формує подію RequestAssigned. Сервіс сповіщень на її основі надсилає користувачу інформацію про реєстрацію та призначення заявки.

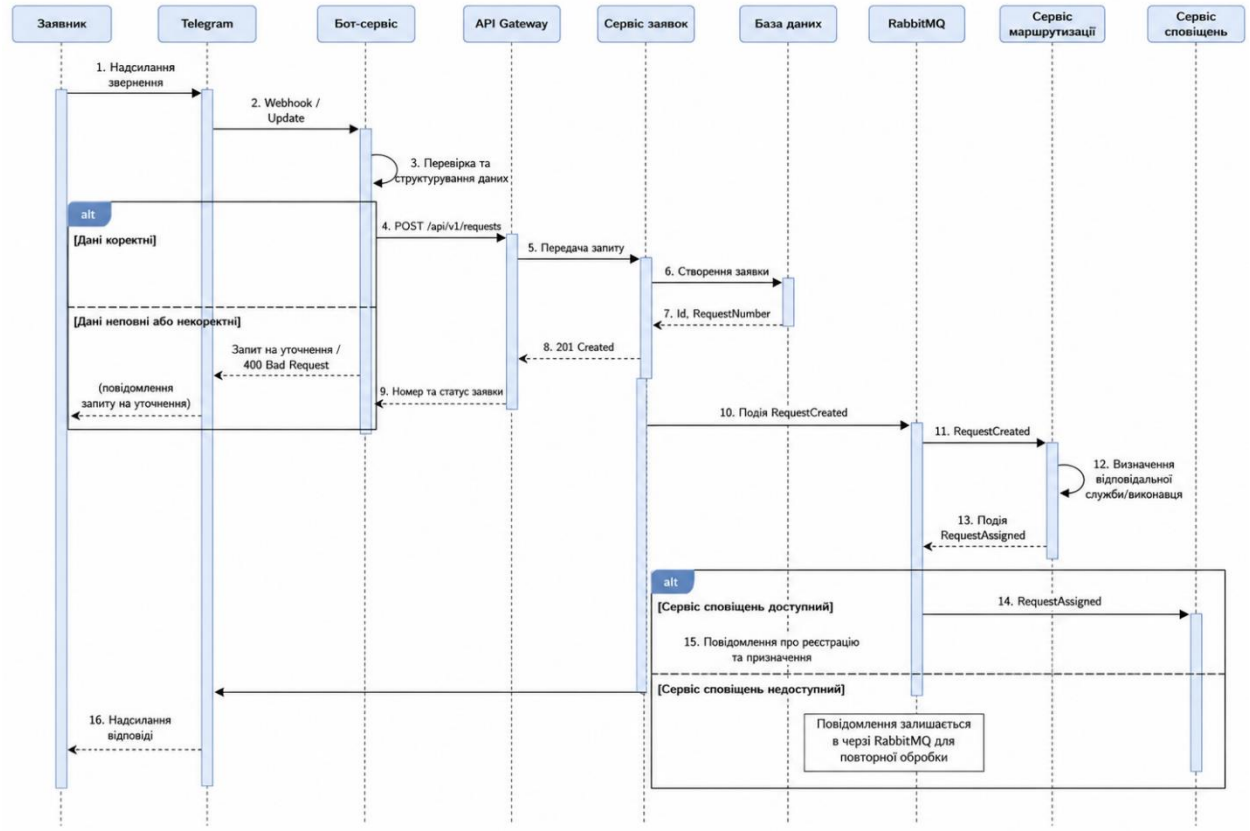

Рисунок 2.4 – UML-діаграма взаємодії сервісів під час створення аварійної заявки

Кожна подія повинна містити унікальний ідентифікатор, тип, час створення, ідентифікатор заявки та корисне навантаження. Це дає змогу відстежувати її обробку та запобігати повторному виконанню однієї операції.

Для захисту API використовується автентифікація на основі JWT-токенів і рольове розмежування доступу. Заявник може працювати лише зі своїми зверненнями, виконавець — із призначеними заявками, а оператор і адміністратор мають розширені права.

Синхронна взаємодія застосовується для операцій, під час яких клієнт очікує негайну відповідь, наприклад створення заявки, перегляду її даних або зміни статусу. Асинхронний обмін через RabbitMQ використовується для маршрутизації, сповіщень і передавання подій до аналітичного сервісу [14]. Це дозволяє уникнути прямої залежності між компонентами та зберегти повідомлення в черзі у разі тимчасової недоступності одержувача.

Основні події асинхронної взаємодії наведено в табл. 2.10.

Таблиця 2.10 – Події взаємодії між мікросервісами

Подія	Джерело	Одержувачі	Призначення
RequestCreated	Сервіс заявок	Сервіс маршрутизації, аналітичний сервіс	Обробка нової заявки
RequestAssigned	Сервіс маршрутизації	Сервіс заявок, сервіс сповіщень	Фіксація призначення
RequestStatusChanged	Сервіс заявок	Сервіс сповіщень, аналітичний сервіс	Обробка зміни статусу
NotificationRequired	Внутрішні сервіси	Сервіс сповіщень	Формування повідомлення
RequestClosed	Сервіс заявок	Аналітичний сервіс	Оновлення статистики

У відповідях API застосовуються стандартні коди HTTP: 200 OK, 201 Created, 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found та 500 Internal Server Error. Для спрощення інтеграції необхідно передбачити єдиний формат помилок, перевірку вхідних даних, журналювання запитів і версіонування маршрутів.

Документування API виконується засобами OpenAPI та Swagger UI. Документація повинна містити маршрути, параметри, моделі запитів і відповідей, коди стану та правила авторизації.

Отже, спроєктований API поєднує REST-взаємодію для синхронних операцій та RabbitMQ для асинхронних подій. Такий підхід забезпечує надійну взаємодію мікросервісів, зменшує їх взаємозалежність і створює умови для подальшого масштабування системи.

2.6 Проєктування механізму інтеграції з месенджером

Інтеграція з месенджером призначена для приймання аварійних заявок, передавання додаткових матеріалів і надсилання користувачу повідомлень про стан звернення. Як основний канал взаємодії обрано Telegram, оскільки Telegram Bot API підтримує текстові повідомлення, команди, інтерактивні кнопки, фотографії, файли та геолокаційні дані [2].

Заявник взаємодіє із системою через клієнтський застосунок Telegram. Платформа месенджера передає події до окремого бот-сервісу через Telegram Bot API. Прямий доступ користувача до внутрішніх API або бази даних не передбачається. Бот-сервіс виконує роль адаптера між зовнішнім каналом комунікації та мікросервісами системи.

Для отримання нових повідомлень доцільно використати механізм webhook. Після надходження команди, тексту, фотографії або геолокації Telegram надсилає HTTPS-запит на адресу бот-сервісу. Це забезпечує оперативну обробку подій без постійного опитування серверів месенджера [2]. Загальну схему інтеграції Telegram-бота з внутрішніми компонентами системи наведено на рис. 2.5.

Як показано на рис. 2.5, вхідний потік проходить через клієнт Telegram, Telegram Bot API, webhook, бот-сервіс та API Gateway до сервісу заявок. Зворотне інформування виконується через сервіс сповіщень і бот-сервіс із подальшим надсиланням повідомлення користувачу через Telegram Bot API.

Основними функціями бот-сервісу є:

– приймання та розпізнавання команд;

– керування етапами діалогу;

– перевірка повноти введених даних;

– обробка фотографій, файлів і геолокації;

– формування структурованого запиту до сервісу заявок;

– надсилання відповідей і статусних повідомлень.

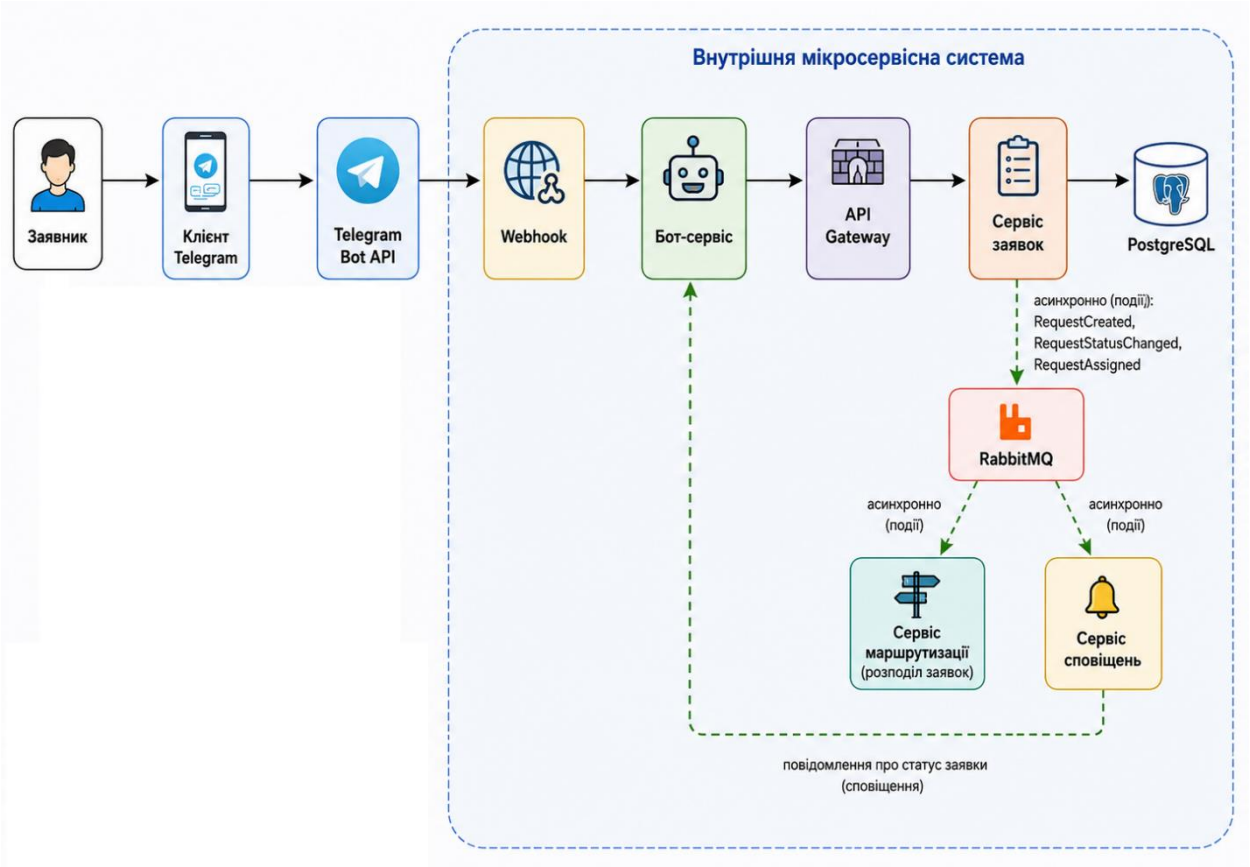

Рисунок 2.5 – Схема інтеграції Telegram-бота з мікросервісною системою обробки заявок

Створення заявки відбувається у вигляді послідовного діалогу. Після команди /start користувач отримує головне меню. Під час вибору функції створення заявки бот послідовно запитує тип аварії, адресу, опис проблеми та додаткові матеріали. Стан діалогу зберігається бот-сервісом, що дає змогу визначати поточний етап введення інформації.

Перед передаванням даних виконується їх валідація. Обов’язковими є тип аварії, адреса або координати та опис проблеми. Якщо дані відсутні або мають некоректний формат, бот надсилає повідомлення про помилку та повторно запитує відповідне значення. Після підтвердження сформована заявка передається через API Gateway до сервісу заявок запитом POST /api/v1/requests.

До запиту можуть входити ідентифікатор користувача Telegram, тип аварійної ситуації, адреса, координати, опис проблеми, ідентифікатори

вкладень, дата та час надходження повідомлення. Після створення запису сервіс заявок повертає номер і початковий статус звернення, які бот-сервіс перетворює на повідомлення для користувача.

Для подальшого інформування застосовується асинхронна взаємодія. Після створення, призначення або зміни статусу заявки відповідний сервіс публікує подію в RabbitMQ [14]. Сервіс сповіщень отримує її, формує текст повідомлення та передає дані бот-сервісу. Користувач може отримувати підтвердження реєстрації, номер заявки, інформацію про призначення виконавця, зміну статусу, запит на уточнення або повідомлення про завершення робіт.

Для перегляду поточного стану звернень бот може містити команду або кнопку «Мої заявки». За ідентифікатором Telegram бот-сервіс отримує список звернень і відображає їх номер, категорію, дату створення та статус.

Під час інтеграції необхідно забезпечити захист даних. Обмін із Telegram і серверною частиною повинен виконуватися через HTTPS. Токен бота слід зберігати у змінних середовища або захищеному сховищі конфігурації. Вхідні webhook-запити мають перевірятися, а доступ бот-сервісу до внутрішніх API — обмежуватися відповідними правами.

Для підвищення надійності доцільно передбачити журналювання подій, повторні спроби надсилання повідомлень і захист від повторної обробки одного оновлення. У разі тимчасової недоступності внутрішнього сервісу помилка фіксується, а користувачу надсилається повідомлення про неможливість виконання операції.

Отже, бот-сервіс забезпечує перетворення повідомлень Telegram у внутрішній формат системи та виконує зворотне інформування заявника. Основна бізнес-логіка залишається в сервісах заявок, маршрутизації та сповіщень, що зменшує залежність системи від конкретного месенджера та спрощує підключення інших каналів взаємодії.

2.7 Забезпечення безпеки та надійності системи

Система обробки аварійних заявок працює з персональними та службовими даними: контактною інформацією, адресами, геолокацією, фотографіями й історією виконання звернень. Тому під час проєктування необхідно забезпечити конфіденційність, цілісність і доступність інформації. Захист системи доцільно реалізувати на декількох рівнях: користувацькому, мережевому, прикладному, рівні даних та інфраструктури. Загальну схему механізмів безпеки й надійності наведено на рис. 2.6.

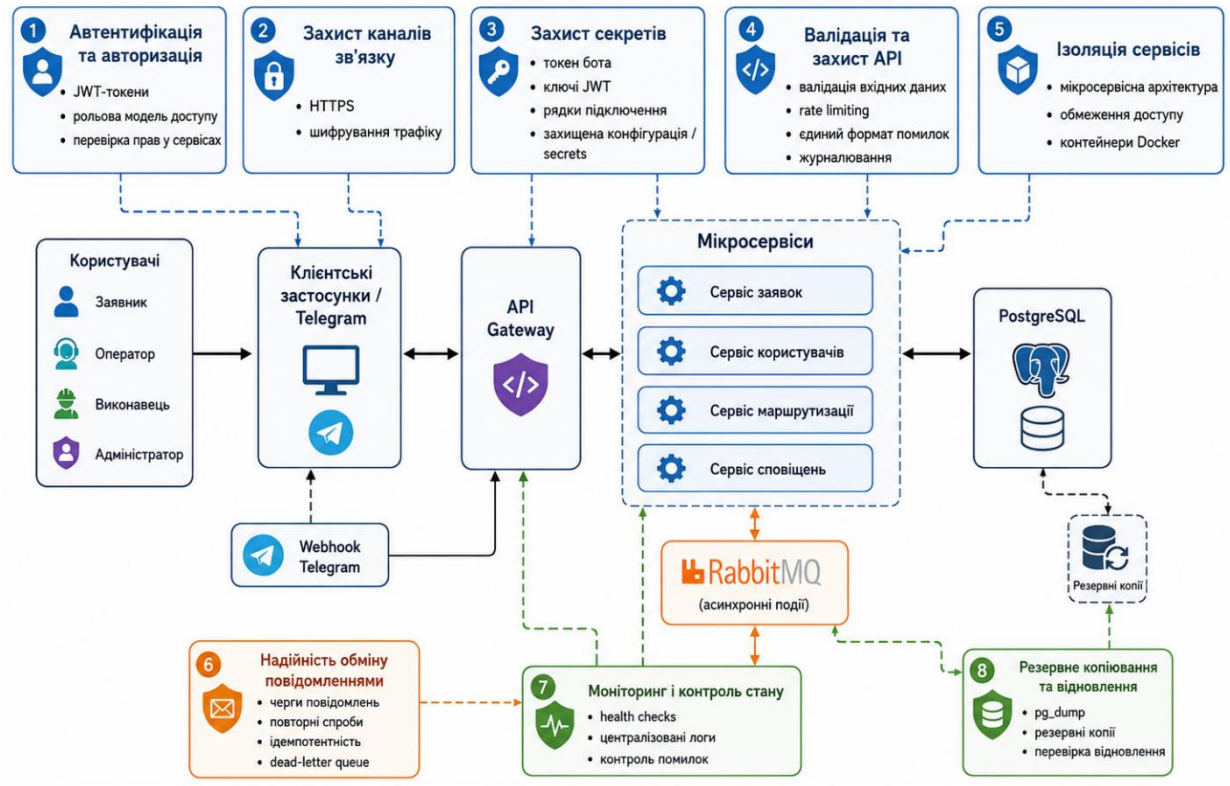

Рисунок 2.6 – Механізми забезпечення безпеки та надійності системи

Доступ до системи здійснюється після автентифікації користувача. Для внутрішніх користувачів передбачено використання JWT-токенів, а права визначаються відповідно до ролі: заявник, оператор, виконавець або адміністратор. Заявник може переглядати лише власні звернення, виконавець — призначені йому заявки, оператор — опрацьовувати та розподіляти звернення, а адміністратор — керувати користувачами й довідниками.

Перевірка прав повинна виконуватися не лише в інтерфейсі, а й безпосередньо в кожному сервісі [20].

Взаємодія клієнтів із системою та обмін даними між сервісами мають здійснюватися через HTTPS. Токен Telegram-бота, ключі JWT і рядки підключення до бази даних не повинні зберігатися у вихідному коді. Їх доцільно передавати через захищену конфігурацію або секрети контейнерного середовища.

Усі вхідні дані повинні проходити валідацію. Система перевіряє обов'язковість полів, формат номера телефону, допустимість координат, розмір тексту, кількість і тип вкладень. Для захисту API також слід застосувати обмеження частоти запитів, єдиний формат помилок і журналювання спроб несанкціонованого доступу. Особливу увагу потрібно приділити перевірці доступу до конкретних заявок, оскільки порушення об'єктної авторизації та помилки автентифікації належать до основних ризиків безпеки API [21].

Бот-сервіс не повинен мати прямого доступу до бази даних. Усі операції виконуються через API Gateway і відповідні внутрішні сервіси. Такий підхід обмежує область доступу компонента, який взаємодіє із зовнішньою платформою. Webhook Telegram має працювати через HTTPS, а отримані події потрібно перевіряти та захищати від повторної обробки.

Надійність системи забезпечується ізоляцією мікросервісів і асинхронним передаванням подій через RabbitMQ. Тимчасова недоступність сервісу сповіщень або аналітики не повинна блокувати створення заявки. Повідомлення зберігається в черзі та обробляється після відновлення відповідного сервісу. Обробники подій мають бути ідемпотентними, щоб повторне отримання повідомлення не спричиняло дублювання операцій.

Для контролю стану компонентів кожен сервіс повинен мати endpoint перевірки працездатності, наприклад /health. Перевіряється доступність сервісу, PostgreSQL, RabbitMQ та інших критичних залежностей. Події створення й зміни заявок, помилки авторизації, збої інтеграції та

адміністративні дії фіксуються в централізованих журналах. При цьому паролі, токени й інші секретні дані не повинні потрапляти до логів.

Захист інформації від втрати передбачає регулярне резервне копіювання PostgreSQL. Для навчального проєкту доцільно виконувати періодичне створення резервних копій за допомогою pg_dump, зберігати їх окремо від основної бази та перевіряти можливість відновлення [22]. Контейнери сервісів слід запускати з мінімальними привілеями та використовувати актуальні образи без зайвих залежностей.

Отже, безпека системи забезпечується автентифікацією, рольовим доступом, шифруванням каналів, валідацією даних, захищеним зберіганням секретів і журналюванням. Надійність підтримується асинхронною обробкою подій, перевіркою стану сервісів, резервним копіюванням та ізоляцією компонентів. Запропоновані заходи відповідають ризик-орієнтованому підходу до захисту інформаційних систем [23].

2.8 Реалізація основних мікросервісів системи

Програмна реалізація системи обробки аварійних заявок виконана відповідно до мікросервісної архітектури, спроєктованої у другому розділі. Функціональність системи розподілено між окремими сервісами, кожен із яких відповідає за визначену предметну область. Такий підхід спрощує супровід програмного коду, дозволяє незалежно тестувати й розгортати компоненти та створює можливість їх подальшого масштабування.

Основні мікросервіси реалізовано на платформі ASP.NET Core у вигляді окремих Web API-застосунків. Кожен сервіс має власну структуру проєкту, що включає контролери, моделі даних, сервіси бізнес-логіки, засоби доступу до бази даних і конфігурацію. Вбудований механізм впровадження залежностей ASP.NET Core використовується для реєстрації репозиторіїв, обробників подій, сервісів валідації та інших компонентів [10].

Сервіс заявок є центральним компонентом системи. Він забезпечує створення, перегляд, оновлення та закриття аварійних звернень. Через REST

API сервіс приймає дані про тип аварії, адресу, опис проблеми, заявника, географічні координати та вкладення.

Під час створення звернення виконується перевірка обов’язкових полів, після чого формується унікальний номер і встановлюється початковий статус «Нова». Дані зберігаються в PostgreSQL за допомогою Entity Framework Core. Після успішного створення заявки сервіс публікує в RabbitMQ подію RequestCreated.

Основними операціями сервісу є:

– створення нової заявки;

– отримання списку та детальної інформації про звернення;

– зміна статусу і пріоритету;

– призначення служби або виконавця;

– додавання вкладень і службових коментарів;

– отримання історії обробки заявки.

Кожна зміна статусу фіксується в таблиці RequestHistory. Це дозволяє відновити послідовність виконання звернення та контролювати дії користувачів.

Сервіс користувачів відповідає за облікові записи заявників, операторів, виконавців та адміністраторів. Він забезпечує збереження профілів, визначення ролей і перевірку прав доступу до функцій системи.

Для внутрішніх користувачів реалізовано автентифікацію з формуванням JWT-токена. Токен містить ідентифікатор користувача та його роль, на основі яких сервіси приймають рішення щодо доступу до операцій. Заявники, які працюють через Telegram, ідентифікуються за значенням MessengerUserId.

Сервіс користувачів підтримує операції реєстрації, пошуку, зміни профілю, блокування облікового запису та отримання переліку виконавців певної служби. Паролі зберігаються у вигляді криптографічних хешів.

Сервіс маршрутизації призначений для визначення відповідальної аварійної служби та виконавця. Після отримання події RequestCreated він аналізує тип аварії, місце події, пріоритет і доступність працівників.

Основою маршрутизації є правила, що пов’язують категорію заявки з відповідною службою. Наприклад, звернення щодо витоку води направляється до служби водопостачання, а повідомлення про пошкодження електромережі — до електротехнічного підрозділу. Якщо автоматичне призначення неможливе, заявка залишається в черзі оператора для ручного розподілу.

Після вибору відповідального виконавця сервіс формує подію RequestAssigned. Сервіс заявок оновлює відповідні поля, а сервіс сповіщень інформує учасників процесу.

Сервіс сповіщень забезпечує формування та надсилання повідомлень заявникам, операторам і виконавцям. Він отримує події RequestCreated, RequestAssigned, RequestStatusChanged і RequestClosed із RabbitMQ.

Після обробки події сервіс визначає одержувача, канал зв’язку та шаблон повідомлення. Для заявників основним каналом є Telegram. До повідомлення можуть входити номер заявки, поточний статус, назва відповідальної служби та результат виконання.

Інформація про кожну спробу надсилання зберігається в таблиці Notifications. У разі помилки статус доставки змінюється, а повідомлення може бути повторно оброблене. Використання RabbitMQ забезпечує асинхронність і не блокує основний процес створення або оновлення заявки [14].

Бот-сервіс реалізує взаємодію з Telegram Bot API та виконує роль адаптера між месенджером і внутрішніми сервісами. Він приймає webhook-події, обробляє команди користувача, керує станом діалогу та формує запити до API Gateway.

Під час створення заявки бот послідовно запитує категорію аварії, адресу, опис, фотографію або геолокацію. Після перевірки дані

перетворюються на модель запиту та передаються до сервісу заявок. У відповідь користувач отримує номер і початковий статус звернення.

Бот також підтримує перегляд раніше створених заявок і отримання їх поточного стану. Основна бізнес-логіка при цьому не дублюється в бот-сервісі, а залишається у відповідних мікросервісах.

API Gateway реалізовано як єдину точку доступу до внутрішніх сервісів. Він приймає запити від бот-сервісу та вебінтерфейсу оператора, після чого маршрутизує їх до сервісу заявок, користувачів або інших компонентів.

На рівні шлюзу виконуються перевірка JWT-токена, журналювання, маршрутизація та базове обмеження частоти запитів. Для реалізації використано YARP, який підтримує конфігурацію маршрутів і кластерів цільових сервісів [13].

Синхронний обмін між компонентами здійснюється через REST API у форматі JSON. Асинхронні операції реалізовано через RabbitMQ. Така комбінація дозволяє використовувати прямі запити для операцій, що потребують негайної відповіді, та події для маршрутизації, сповіщень і аналітики.

Доступ до PostgreSQL реалізовано засобами Entity Framework Core. Для створення та оновлення структури таблиць використовуються міграції [11]. У навчальній реалізації сервіси працюють з однією фізичною базою даних, але використовують логічно відокремлені таблиці.

Кожен компонент упаковано в окремий Docker-контейнер. Конфігурація Docker Compose описує мікросервіси, PostgreSQL, RabbitMQ, мережі, порти та змінні середовища [15]. Це дає змогу запускати всю систему однією командою та забезпечує однакове середовище виконання під час розробки й тестування.

Отже, у межах реалізації створено набір взаємопов’язаних мікросервісів, що забезпечують повний цикл роботи з аварійними заявками. Поділ функціональності між сервісами заявок, користувачів, маршрутизації, сповіщень і Telegram-ботом відповідає спроєктованій архітектурі та створює основу для подальшого тестування і практичної апробації системи.

2.9 Реалізація інтерфейсу оператора або адміністратора

Для керування аварійними заявками реалізовано вебінтерфейс оператора та адміністратора. Він забезпечує перегляд і опрацювання звернень, призначення відповідальних виконавців, зміну статусів, контроль строків виконання, а також керування користувачами та довідниками системи. Інтерфейс побудовано засобами ASP.NET Core MVC і взаємодіє з внутрішніми сервісами через API Gateway. Архітектурний шаблон MVC розділяє застосунок на моделі, представлення та контролери, що спрощує організацію програмного коду, тестування й подальший супровід вебзастосунку [24].

Після автентифікації користувач переходить до головної панелі. Набір доступних функцій залежить від його ролі. Оператор має доступ до реєстру заявок, фільтрації, перегляду детальної інформації, призначення виконавців і зміни статусів. Адміністратор додатково може керувати обліковими записами, ролями, службами, типами аварій і системними довідниками. Розмежування доступу реалізовано на основі JWT-токенів і рольової авторизації [20]. ASP.NET Core дозволяє обмежувати доступ до контролерів і окремих дій за допомогою ролей та політик авторизації [25]. Завдяки цьому оператор не може відкрити адміністративні сторінки навіть шляхом прямого переходу за їх адресою.

Головна панель містить узагальнені показники роботи системи: загальну кількість заявок, кількість нових, активних, прострочених і завершених звернень. Також відображаються останні заявки та статистика їх розподілу за статусами. Приклад головної панелі оператора наведено на рис. 2.7.

Основним робочим елементом є реєстр заявок. У таблиці відображаються номер звернення, адреса, тип аварії, пріоритет, статус, відповідальна служба та дата створення. Передбачено пошук за номером або адресою, а також фільтрацію за статусом, категорією, пріоритетом, службою і часовим періодом. Заявки з високим пріоритетом або перевищеним строком виконання виділяються візуально. Інтерфейс списку заявок показано на рис. 2.8.

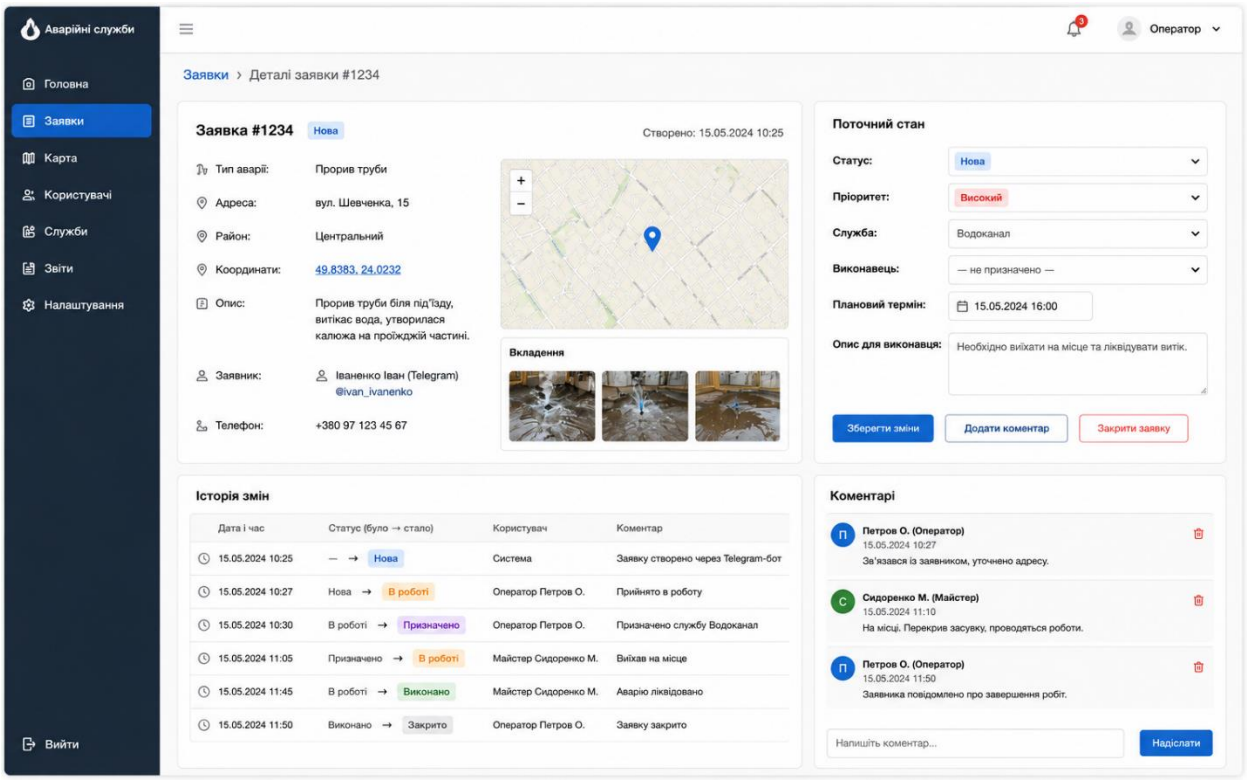

Рисунок 2.7 – Головна панель оператора системи обробки аварійних заявок

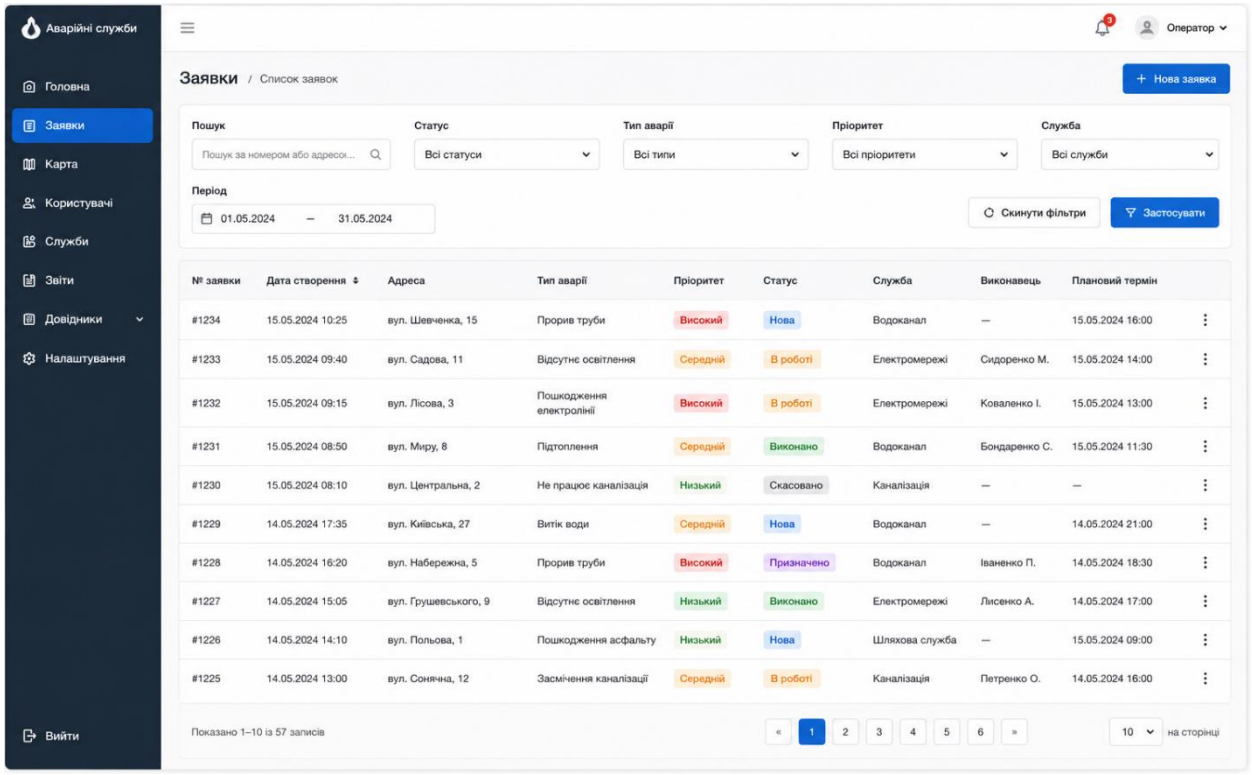

Рисунок 2.8 – Інтерфейс перегляду та фільтрації аварійних заявок

Після вибору запису відкривається картка заявки. Вона містить основні дані про звернення: номер, дату створення, відомості про заявника, тип аварії, адресу, координати, опис проблеми, фотографії, пріоритет, статус і призначеного виконавця. Оператор може змінити пріоритет, призначити службу або працівника, додати службовий коментар та виконати допустимий перехід між статусами.

Окремий блок картки містить історію опрацювання заявки. У ньому відображаються час події, попередній і новий статус, користувач, який виконав зміну, та коментар. Після оновлення даних сервіс заявок створює запис у таблиці RequestHistory і формує подію RequestStatusChanged. Приклад детального перегляду звернення наведено на рис. 2.9.

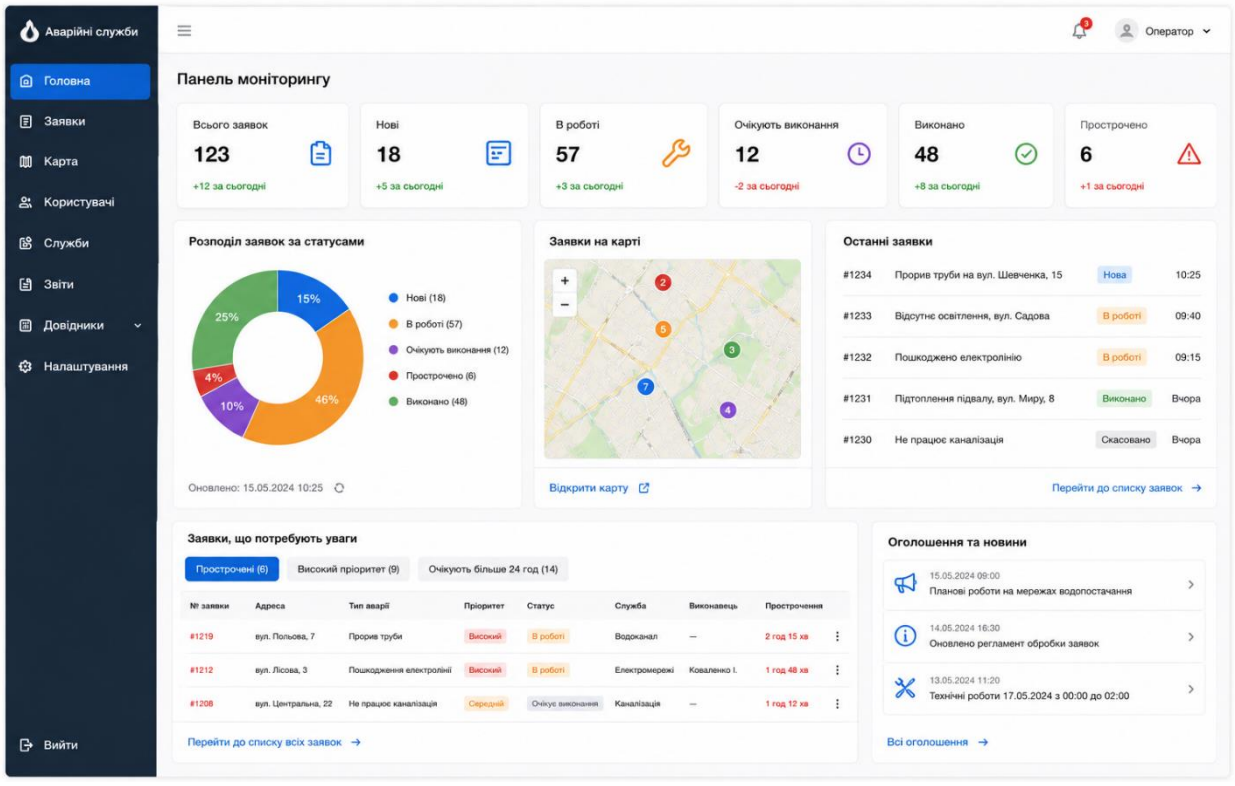

Рисунок 2.9 – Інтерфейс перегляду та опрацювання аварійної заявки

Для критичних операцій, зокрема закриття, скасування або перепризначення заявки, передбачено додаткове підтвердження. Результат виконання операції відображається у вигляді інформаційного повідомлення.

Це знижує ризик випадкової зміни даних і підвищує контрольованість роботи оператора. Під час реалізації таблиці враховано основні сценарії роботи з великими наборами даних: пошук потрібного запису, фільтрацію, порівняння показників, перегляд окремого рядка та виконання дій над ним [26].

Адміністративна частина використовується для керування користувачами та довідниками. На сторінці користувачів відображаються ім’я, роль, контактні дані, служба та стан облікового запису. Адміністратор може створювати нові облікові записи, змінювати ролі, прив’язувати працівників до служб та блокувати їх доступ.

Окремі сторінки передбачені для керування аварійними службами, типами звернень і статусами. Для кожного типу аварії можна задати початковий пріоритет і службу, що використовується під час автоматичної маршрутизації. Це дозволяє змінювати правила роботи системи без редагування програмного коду.

Для наочного подання логіки навігації на рис. 2.10 наведено UML-діаграму переходів між основними інтерфейсами системи.

Після входу користувач переходить до головної панелі, звідки може відкрити список заявок, картку конкретного звернення, розділ звітів або доступні адміністративні сторінки. Після завершення операції користувач повертається до попереднього розділу або головної панелі. Доступ до адміністративних сторінок надається лише користувачам із відповідною роллю.

Інтерфейс має адаптивне компонування та може використовуватися на настільних комп’ютерах і планшетах. Структура API та моделей запитів документується засобами OpenAPI, що спрощує інтеграцію інтерфейсу з мікросервісами. Для підвищення доступності інтерфейсу використано зрозумілі підписи елементів, достатній контраст і текстове дублювання кольорових статусів. Такі рішення відповідають загальним рекомендаціям WCAG 2.2 щодо сприйнятності, керованості та зрозумілості вебінтерфейсів [27].

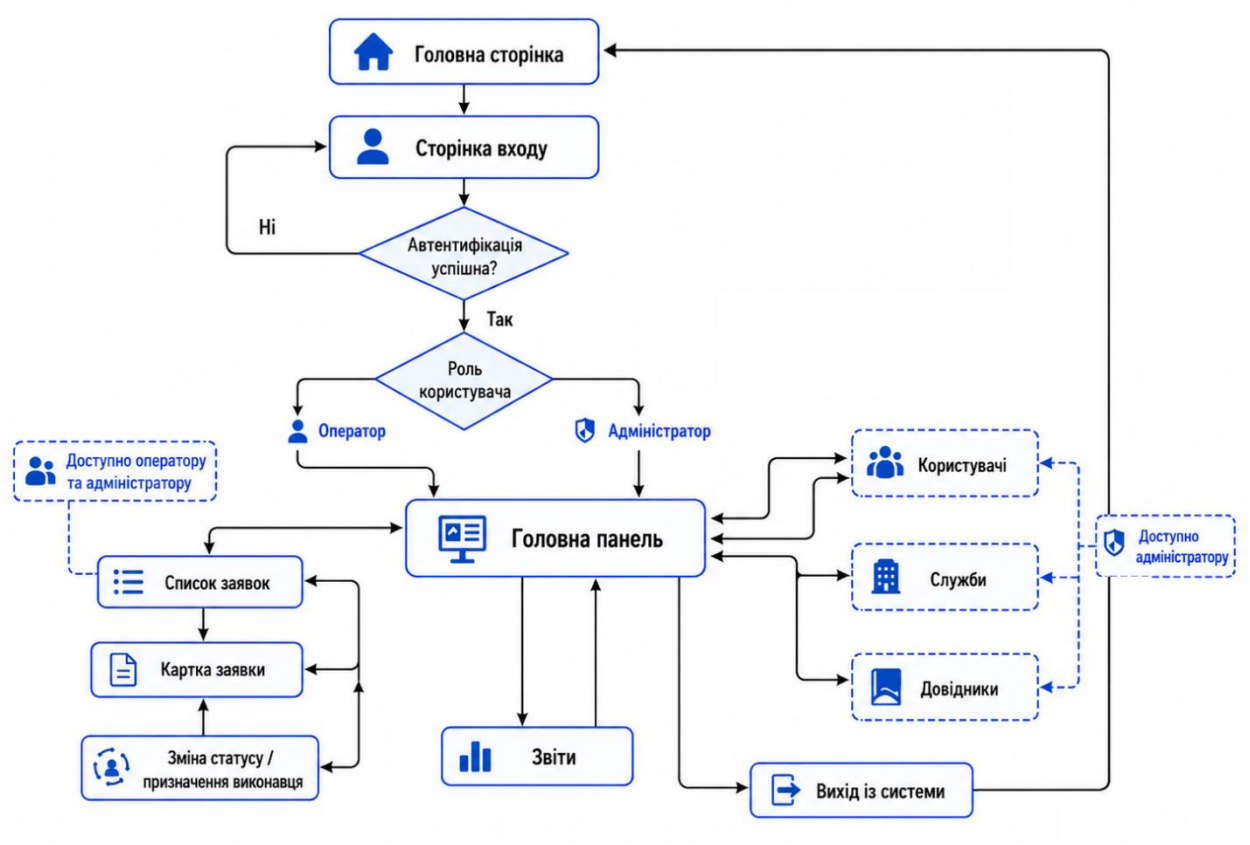

Рисунок 2.10 – UML-діаграма переходів між основними інтерфейсами системи

Отже, реалізований вебінтерфейс забезпечує оператору централізовану роботу із заявками, а адміністратору — керування користувачами, службами та довідниками. Фільтрація, історія змін, візуальне подання статусів і рольове розмежування доступу підвищують зручність та прозорість процесу обробки звернень.

2.10 Реалізація Telegram-бота для створення заявок

Telegram-бот реалізовано як окремий сервіс, що забезпечує взаємодію заявника з мікросервісною системою. Його основними завданнями є приймання повідомлень, керування діалогом, перевірка введених даних, передавання заявки до серверної частини та інформування користувача про результат. Telegram Bot API підтримує команди, кнопки, текстові

повідомлення, фотографії, файли та геолокацію, що відповідає вимогам системи аварійних звернень [2, 28].

Бот створюється та налаштовується через BotFather, після чого отриманий токен зберігається у захищеній конфігурації. Для приймання подій використано webhook: платформа Telegram надсилає HTTPS-запити до бот-сервісу після отримання повідомлення або натискання кнопки. Порівняно з постійним опитуванням API такий механізм забезпечує оперативне надходження подій і краще відповідає серверному розгортанню [28].

Після команди /start користувачу відображається головне меню з основними діями: «Створити заявку», «Мої заявки» та «Допомога». Під час створення звернення бот послідовно запитує тип аварії, адресу або геолокацію, опис проблеми та, за потреби, фотографію. Використання кнопок зменшує кількість помилок під час вибору категорії та спрощує роботу користувача.

Для керування діалогом застосовується модель станів. Для кожного користувача зберігається поточний етап заповнення заявки та вже введені значення. Основні стани включають вибір категорії, введення адреси, опис проблеми, додавання вкладення та підтвердження. Команда скасування припиняє сценарій і видаляє проміжні дані.

Перед створенням заявки бот перевіряє наявність обов’язкових полів і допустимість отриманих значень. Фотографії не передаються безпосередньо в основній моделі заявки: бот отримує ідентифікатор файлу Telegram, завантажує його або формує посилання для подальшого збереження. Геолокація перетворюється на координати широти й довготи.

Після підтвердження бот-сервіс формує JSON-запит і надсилає його через API Gateway до маршруту POST /api/v1/requests. Запит містить ідентифікатор користувача Telegram, категорію аварії, адресу, координати, опис та відомості про вкладення. У разі успішного створення сервіс заявок повертає код 201 Created, номер і початковий статус звернення.

Схему взаємодії компонентів під час створення заявки через Telegram-бот наведено на рис. 2.11.

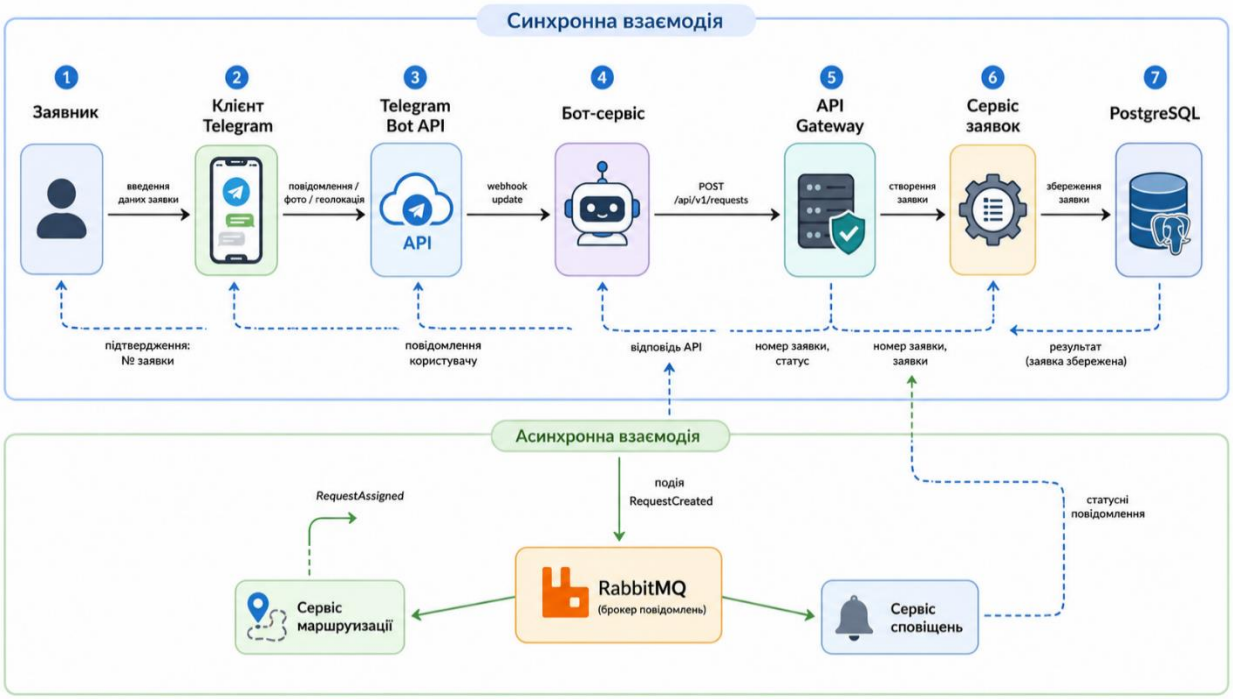

Рисунок 2.11 – Схема взаємодії компонентів під час створення заявки через Telegram-бот

Як показано на рис. 2.11, заявник працює через клієнт Telegram. Платформа Telegram передає оновлення бот-сервісу, який перевіряє та структурує дані. Далі запит через API Gateway надходить до сервісу заявок і зберігається в PostgreSQL. Після створення заявки публікується подія RequestCreated, яку обробляють сервіси маршрутизації та сповіщень через RabbitMQ.

Для перегляду стану звернень реалізовано функцію «Мої заявки». Бот отримує список заявок користувача та відображає їх номер, дату створення, категорію і поточний статус. Повідомлення про призначення виконавця, зміну статусу та завершення робіт надсилаються автоматично сервісом сповіщень.

Під час обробки помилок користувач отримує зрозуміле повідомлення без розкриття внутрішніх деталей системи. Помилки API, недоступність Telegram та невдалі спроби надсилання фіксуються в журналі. Для тимчасових збоїв застосовуються повторні спроби, а події сповіщень зберігаються в черзі RabbitMQ [14].

Отже, Telegram-бот забезпечує зручний сценарій створення аварійної заявки, приймання мультимедійних матеріалів і подальший зворотний зв’язок. Винесення інтеграції в окремий сервіс запобігає дублюванню бізнес-логіки та дозволяє надалі підключати інші канали взаємодії.

2.11 Реалізація обміну повідомленнями між мікросервісами

Для обміну подіями між мікросервісами реалізовано асинхронну взаємодію через брокер повідомлень RabbitMQ. На відміну від синхронних HTTP-запитів, цей підхід не потребує одночасної доступності сервісу-відправника та сервісу-одержувача. Видавець формує подію та передає її брокеру, після чого відповідні сервіси отримують і обробляють повідомлення незалежно один від одного. Подієва взаємодія зменшує зв’язаність мікросервісів і дає змогу підключати нових споживачів без зміни сервісу-видавця [29].

У системі сервіс заявок є основним видавцем подій. Після створення або зміни заявки він формує повідомлення, що містить тип події, ідентифікатор заявки, час створення та необхідні дані для подальшої обробки. Основними інтеграційними подіями є:

– RequestCreated — створено нову заявку;

– RequestAssigned — визначено службу або виконавця;

– RequestStatusChanged — змінено статус заявки;

– RequestClosed — заявку закрито;

– NotificationRequired — потрібно надіслати повідомлення користувачу.

Повідомлення передається до RabbitMQ exchange, який маршрутизує його до однієї або декількох черг відповідно до routing key. У RabbitMQ видавці надсилають повідомлення до exchange, а той на основі типу та налаштованих зв’язків направляє їх до потрібних черг [30]. Для системи передбачено окремі черги маршрутизації, сповіщень та аналітики.

Схему обміну подіями та основні інформаційні потоки між сервісами наведено на рис. 2.12.

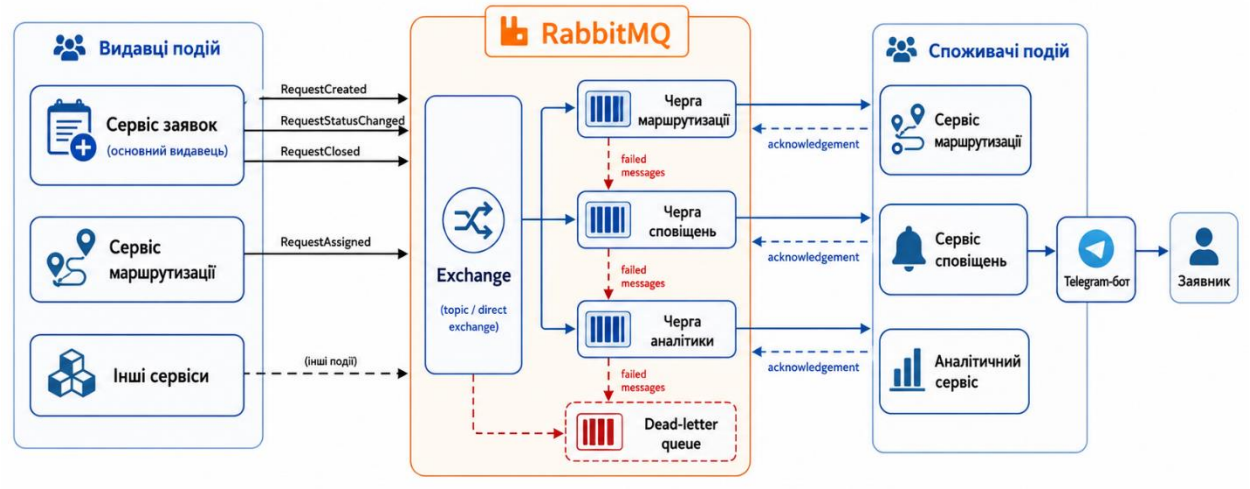

Рисунок 2.12 – Схема асинхронного обміну повідомленнями між мікросервісами

Як показано на рис. 2.12, після створення заявки сервіс заявок публікує подію RequestCreated. Сервіс маршрутизації отримує її, визначає відповідальну службу та формує подію RequestAssigned. Сервіс сповіщень обробляє події призначення і зміни статусу, після чого надсилає повідомлення заявнику через Telegram-бот. Аналітичний сервіс використовує ці події для оновлення статистичних показників.

Структуру інтеграційної події реалізовано у вигляді JSON-повідомлення. Узагальнений формат містить такі поля:

```
{
  "eventId": "2ad9a667-139c-4e72-82fe-11b0a390fa20",
  "eventType": "RequestStatusChanged",
  "occurredAt": "2026-06-07T10:25:00Z",
  "requestId": "174da222-732e-4fd2-836d-a518357360ab",
  "payload": {
    "previousStatus": "Assigned",
    "newStatus": "InProgress",
    "changedByUserId": "8041c724-dfcf-4ce5-8077-54388a06ed32"
  }
}
```

Поле eventId використовується для ідентифікації повідомлення та захисту від повторної обробки. eventType визначає тип події, occurredAt — час її формування, а payload містить дані, потрібні конкретним споживачам.

Після отримання повідомлення сервіс виконує необхідну операцію та надсилає підтвердження RabbitMQ. Consumer acknowledgement повідомляє брокеру, що подію успішно опрацьовано і її можна видалити з черги. Для контролю надсилання видавцем застосовуються publisher confirms [14]. Якщо споживач завершує роботу до підтвердження, брокер може повторно доставити повідомлення.

Через можливість повторної доставки обробники реалізовано ідемпотентними. Перед виконанням операції сервіс перевіряє eventId у журналі оброблених подій. Якщо повідомлення з таким ідентифікатором уже опрацьовано, повторна зміна даних або надсилання однакового сповіщення не виконується.

Для тимчасових помилок застосовуються повторні спроби з інтервалом між ними. Повідомлення, яке не вдалося обробити після встановленої кількості спроб, направляється до dead-letter queue. Це запобігає блокуванню основної черги та дає змогу окремо проаналізувати причину помилки.

Параметри підключення до RabbitMQ, назви exchange і черг задаються в конфігурації сервісів. Під час запуску кожний споживач створює необхідні черги, зв’язки та підписки. У Docker Compose брокер запускається як окремий контейнер, доступний мікросервісам у внутрішній мережі.

Отже, реалізований механізм обміну повідомленнями забезпечує незалежність сервісів, надійну доставку подій і можливість асинхронної обробки заявок. Використання підтверджень, повторних спроб, ідемпотентних обробників і dead-letter queue підвищує стійкість системи до тимчасових помилок та часткових відмов.

2.12 Тестування системи

Тестування розробленої системи виконувалося з метою перевірки коректності бізнес-логіки, взаємодії мікросервісів, роботи REST API, Telegram-бота та механізму асинхронного обміну повідомленнями. Перевірка охоплювала модульний, інтеграційний, функціональний і системний рівні. Модульні тести використовувалися для ізольованої перевірки окремих класів і методів, тоді як інтеграційні тести підтверджували спільну роботу декількох компонентів [31].

Для автоматизованого тестування серверної частини застосовано фреймворк xUnit, який підтримує створення тестових методів, параметризованих сценаріїв і групування перевірок у проєктах .NET [32]. Тести виконуються окремо від основного застосунку та не впливають на робочі дані системи.

Модульне тестування охоплювало основні правила обробки заявки:

– перевірку обов’язкових полів;

– формування унікального номера звернення;

– визначення початкового статусу;

– обчислення пріоритету;

– перевірку допустимих переходів між статусами;

– вибір служби за типом аварії;

– формування тексту сповіщення.

Наприклад, під час створення заявки без опису або адреси сервіс повинен повернути помилку валідації. Перехід зі статусу «Нова» до «Призначена» є допустимим, тоді як безпосередній перехід до стану «Закрита» повинен бути відхилений.

Інтеграційне тестування використовувалося для перевірки REST API, доступу до PostgreSQL і взаємодії з RabbitMQ. Тестовий вебсервер приймав HTTP-запити до маршрутів сервісу заявок і повертав відповіді без необхідності ручного запуску клієнтського інтерфейсу. Такий підхід дозволяє

перевірити маршрутизацію, серіалізацію JSON, валідацію, авторизацію та збереження даних [31].

Для відтворення середовища PostgreSQL і RabbitMQ можуть використовуватися тимчасові Docker-контейнери, які створюються перед запуском тестів і видаляються після їх завершення. Testcontainers for .NET надає засоби програмного запуску ізольованих контейнерів із необхідними залежностями [33]. Завдяки цьому інтеграційні тести виконуються в умовах, наближених до реального розгортання.

Основні тестові сценарії наведено в табл. 2.11.

Таблиця 2.11 – Результати тестування основних функцій системи

№	Тестовий сценарій	Очікуваний результат	Результат
1	Створення заявки з коректними даними	Код 201 Created, сформовано номер заявки	Успішно
2	Створення заявки без обов’язкового опису	Код 400 Bad Request	Успішно
3	Отримання заявки за ідентифікатором	Повернено повні дані звернення	Успішно
4	Запит неіснуючої заявки	Код 404 Not Found	Успішно
5	Зміна статусу за допустимим переходом	Статус оновлено, створено запис історії	Успішно
6	Недопустимий перехід між статусами	Операцію відхилено	Успішно
7	Призначення відповідальної служби	Оновлено заявку, створено RequestAssigned	Успішно
8	Публікація події в RabbitMQ	Повідомлення передано до відповідної черги	Успішно
9	Повторне отримання однакової події	Повторна операція не виконується	Успішно
10	Доступ без чинного JWT-токена	Код 401 Unauthorized	Успішно
11	Доступ оператора до адміністративної функції	Код 403 Forbidden	Успішно
12	Створення заявки через Telegram-бот	Користувач отримує номер і статус	Успішно

Окремо перевірено роботу Telegram-бота. Під час функціонального тестування виконувалися сценарії початку діалогу, вибору категорії, введення адреси, надсилання геолокації та фотографії, підтвердження заявки, перегляду списку звернень і скасування введення. Якщо користувач надсилав дані неправильного формату, бот повторно запитував потрібну інформацію та не створював неповну заявку.

Тестування обміну повідомленнями охоплювало публікацію подій RequestCreated, RequestAssigned і RequestStatusChanged, їх маршрутизацію до відповідних черг та обробку сервісами-споживачами. Також перевірено повторну доставку повідомлення у разі відсутності підтвердження та перенесення помилкових подій до dead-letter queue.

Перевірка безпеки включала доступ до захищених маршрутів без токена, використання токена з недостатніми правами, передавання некоректних параметрів і перевищення встановленого розміру вкладення. Система повинна повертати відповідні HTTP-коди та не розкривати внутрішні технічні відомості у повідомленнях про помилки.

Результати запуску автоматизованих тестів доцільно подати на рис. 2.13.

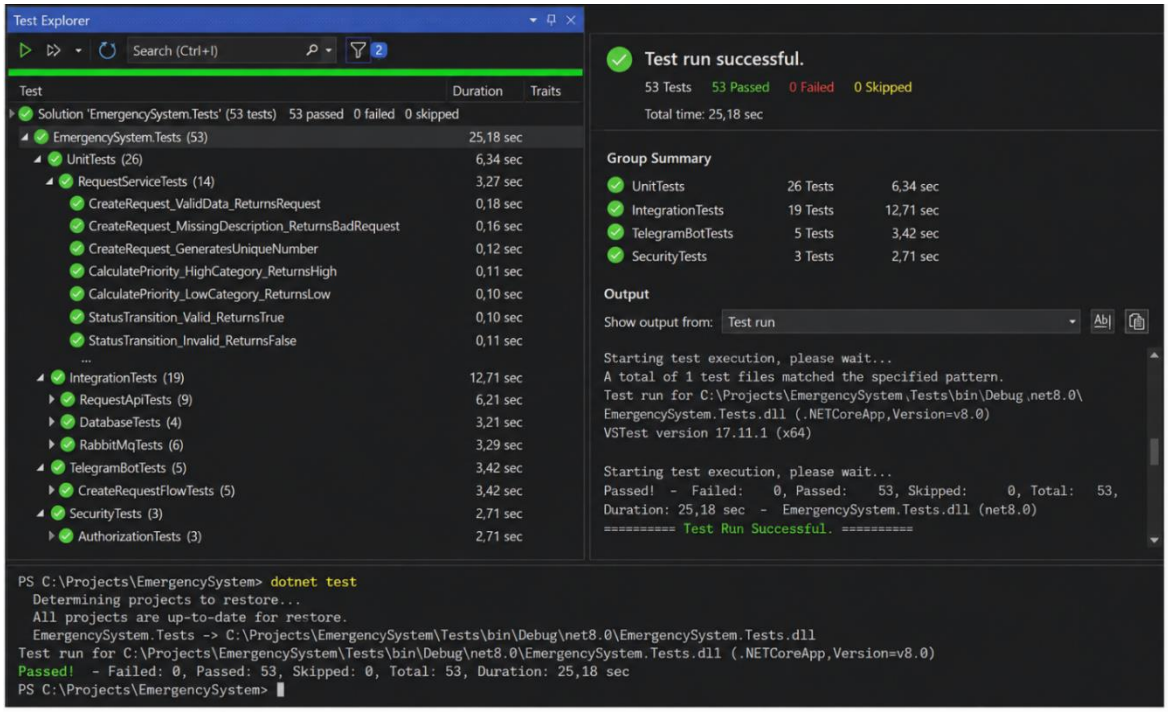

Рисунок 2.13 – Результати виконання автоматизованих тестів системи

На рисунку можуть бути відображені вікно Test Explorer або результат виконання команди dotnet test із кількістю успішних і невдалих перевірок. Доцільно також показати тривалість виконання тестів і назви основних тестових наборів.

За результатами тестування встановлено, що основні функції системи виконуються відповідно до сформованих вимог. Система коректно створює та обробляє заявки, контролює статусні переходи, розмежовує права користувачів, передає події між мікросервісами та забезпечує взаємодію з Telegram-ботом. Виявлені під час перевірки помилки усувалися шляхом уточнення правил валідації, обробки винятків і конфігурації черг повідомлень.

2.13 Практична апробація системи

Практичну апробацію розробленої системи проведено в локальному тестовому середовищі, яке відтворює основні умови її використання аварійною службою. Метою апробації було підтвердження працездатності повного циклу обробки заявки: від надсилання повідомлення заявником через Telegram до виконання робіт, зміни статусу та отримання підсумкового сповіщення.

До складу демонстраційного середовища входили мікросервіси заявок, користувачів, маршрутизації та сповіщень, бот-сервіс, API Gateway, PostgreSQL і RabbitMQ. Компоненти запускалися в окремих Docker-контейнерах за допомогою Docker Compose. Оператор працював із системою через вебінтерфейс, а роль заявника виконувалася через клієнтський застосунок Telegram.

Оцінювання здійснювалося за критеріями функціональної придатності, продуктивності взаємодії, надійності, зручності використання та захищеності. Такі характеристики відповідають моделі якості програмного продукту ISO/IEC 25010 [34]. Для фіксації результатів використовувалися кількість успішно завершених сценаріїв, час виконання операцій, коректність

збереження даних і доставка статусних повідомлень. Застосування кількісних показників узгоджується з принципами оцінювання якості програмних систем ISO/IEC 25023 [35].

Основний сценарій апробації передбачав такі дії:

1. Заявник запускав Telegram-бот і обирав функцію створення заявки.
2. У діалозі зазначалися тип аварії, адреса, опис проблеми, фотографія та геолокація.
3. Бот-сервіс передавав сформовані дані через API Gateway до сервісу заявок.
4. Заявка зберігалася в PostgreSQL і отримувала унікальний номер.
5. Подія RequestCreated передавалася через RabbitMQ до сервісу маршрутизації.
6. Оператор переглядав заявку у вебінтерфейсі, визначав пріоритет і призначав виконавця.
7. Виконавець змінював статус заявки на «У роботі», а після завершення — на «Виконана».
8. Заявник отримував через Telegram повідомлення про реєстрацію, призначення та завершення робіт.

Під час апробації додатково перевірено створення заявки з неповними даними, надсилання фотографії та геолокації, недопустимий перехід між статусами, повторну доставку події RabbitMQ і тимчасову недоступність сервісу сповіщень. У випадку відсутності обов’язкових даних бот повторно запитував необхідну інформацію. Некоректні переходи між статусами відхилялися серверною частиною, а повідомлення, яке не вдалося доставити, залишалося в черзі для повторної обробки.

Результати практичної перевірки наведено в табл. 2.12.

Під час апробації підтверджено узгоджену роботу клієнтських інтерфейсів, API Gateway, бази даних і брокера повідомлень. Заявки коректно зберігалися та відображалися у вебінтерфейсі, зміни фіксувалися в історії, а

повідомлення надходили заявнику через Telegram. Асинхронна взаємодія дозволяла продовжувати основний процес навіть за тимчасової недоступності сервісу сповіщень.

Таблиця 2.12 – Результати практичної апробації системи

Перевірений процес	Критерій успішності	Отриманий результат
Створення заявки через Telegram	Заявка збережена, сформовано номер	Виконано
Передавання фото та геолокації	Дані пов’язані із заявкою	Виконано
Автоматична маршрутизація	Визначено відповідну службу	Виконано
Робота оператора із заявкою	Дані доступні для перегляду й оновлення	Виконано
Зміна статусів	Переходи перевірено, історію збережено	Виконано
Надсилання сповіщень	Заявник отримав статусні повідомлення	Виконано
Повторна обробка подій	Дублювання операцій відсутнє	Виконано
Рольове обмеження доступу	Недозволені функції заблоковано	Виконано

Практична перевірка також показала, що діалоговий сценарій бота спрощує введення даних і зменшує ймовірність створення неповної заявки. Вебінтерфейс забезпечує оператору швидкий доступ до активних звернень, засобів фільтрації, призначення виконавців і контролю статусів.

Отже, результати апробації підтвердили працездатність реалізованої системи та відповідність її основним функціональним вимогам. Програмне рішення забезпечує повний цикл обробки аварійної заявки й може використовуватися як прототип для подальшого впровадження, розширення та перевірки в умовах реальної аварійної або комунальної служби.

2.14 Розгортання системи

Розгортання системи обробки аварійних заявок виконано з використанням контейнеризації. Кожний мікросервіс, база даних PostgreSQL, брокер повідомлень RabbitMQ, API Gateway, бот-сервіс і вебінтерфейс запускаються в окремих Docker-контейнерах. Такий підхід ізолює залежності компонентів, забезпечує однакове середовище виконання та спрощує перенесення системи між комп’ютерами.

Для керування багатоконтейнерним застосунком використано Docker Compose. Конфігураційний файл docker-compose.yml визначає перелік сервісів, образи або правила їх побудови, порти, внутрішні мережі, томи, змінні середовища та залежності запуску. Docker Compose дозволяє запускати всі компоненти системи однією командою та підтримує розгортання багатоконтейнерних застосунків на одному сервері [15, 36].

До складу конфігурації розгортання входять:

– контейнер API Gateway;

– сервіси заявок, користувачів, маршрутизації та сповіщень;

– Telegram-бот;

– вебінтерфейс оператора;

– PostgreSQL;

– RabbitMQ;

– окрема внутрішня мережа Docker;

– постійні томи для збереження даних.

Загальну схему розгортання компонентів наведено на рис. 2.14.

Як показано на рис. 2.14, зовнішні запити надходять через API Gateway, тоді як безпосередній доступ до внутрішніх мікросервісів, PostgreSQL і RabbitMQ обмежено внутрішньою мережею Docker. Telegram взаємодіє з бот-сервісом через HTTPS-webhook, а оператор відкриває вебінтерфейс через браузер.

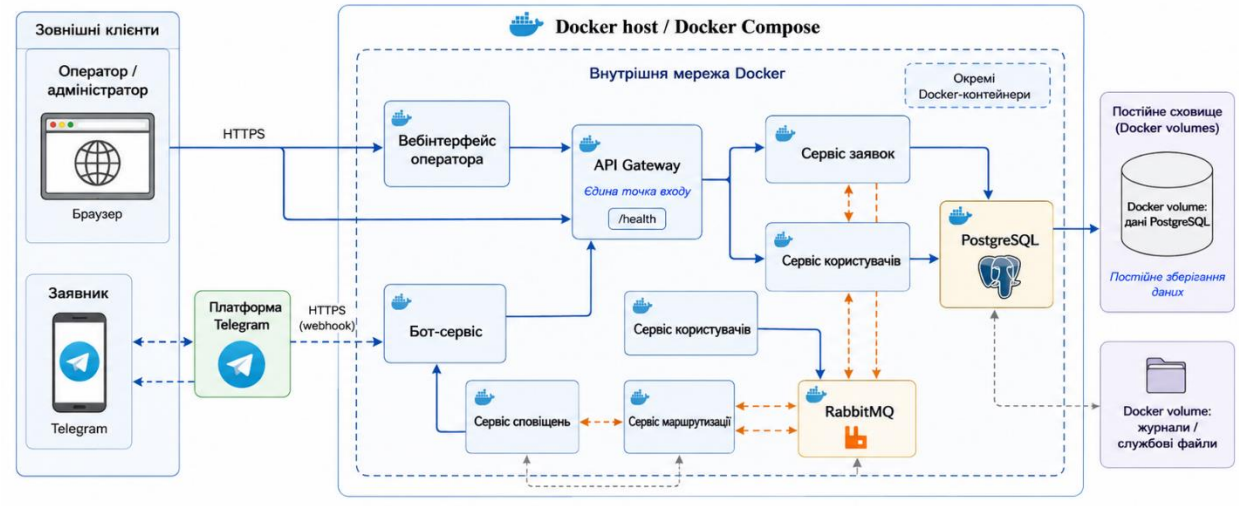

Рисунок 2.14 – Схема контейнерного розгортання системи обробки аварійних заявок

Для PostgreSQL використовується окремий Docker volume. Постійний том зберігає дані незалежно від життєвого циклу контейнера, тому його перезапуск або повторне створення не призводить до втрати записів [37]. Окремий том може застосовуватися для журналів RabbitMQ та службових файлів системи.

Налаштування середовища передаються через змінні конфігурації. До них належать рядок підключення до PostgreSQL, адреса RabbitMQ, JWT-ключ, токен Telegram-бота та адреси внутрішніх сервісів. Конфіденційні значення не зберігаються безпосередньо у вихідному коді або Docker-образах.

Для ASP.NET Core встановлюється середовище Production. Воно призначене для конфігурації застосунку з урахуванням вимог до безпеки, продуктивності та надійності [38]. Параметри середовища розробки та промислового запуску розділяються, що дозволяє використовувати різні адреси сервісів, рівні журналювання й налаштування бази даних.

Порядок розгортання системи включає такі основні етапи:

1. Встановлення Docker Engine і Docker Compose.
2. Підготовку конфігураційних файлів і змінних середовища.
3. Побудову Docker-образів мікросервісів.
4. Запуск контейнерів командою docker compose up -d.

5. Застосування міграцій бази даних.
6. Перевірку стану контейнерів і службових endpoint /health.
7. Налаштування HTTPS-webhook Telegram-бота.
8. Виконання контрольного сценарію створення заявки.

Для забезпечення правильного порядку запуску використовуються залежності між сервісами та перевірки готовності. Сервіс заявок повинен починати повноцінну роботу після доступності PostgreSQL, а сервіси маршрутизації й сповіщень — після запуску RabbitMQ. Політика автоматичного перезапуску дозволяє відновити роботу контейнера після тимчасового збою.

Оновлення системи виконується шляхом побудови нового образу відповідного сервісу та повторного запуску його контейнера. Завдяки функціональному поділу немає необхідності одночасно перевстановлювати всі компоненти. Перед оновленням структури даних створюється резервна копія PostgreSQL, після чого застосовуються міграції Entity Framework Core.

Контроль розгорнутої системи здійснюється за допомогою журналів контейнерів, health checks і стану черг RabbitMQ. Перевіряються доступність API Gateway, з’єднання з базою даних, робота брокера повідомлень та обробка webhook Telegram. У разі помилки адміністратор може переглянути журнал окремого контейнера, не зупиняючи всю систему.

Отже, контейнерне розгортання забезпечує ізоляцію компонентів, повторюваність конфігурації та спрощення запуску системи. Використання Docker Compose, внутрішньої мережі, постійних томів і перевірок працездатності створює зручне середовище для демонстрації, тестування та подальшого перенесення програмного рішення на сервер.

Висновки до розділу:

У розділі виконано проєктування та програмну реалізацію системи обробки заявок для аварійних служб на основі мікросервісної архітектури. Обґрунтовано вибір ASP.NET Core для серверної частини, Entity Framework

Core для доступу до даних, PostgreSQL для їх зберігання, RabbitMQ для асинхронного обміну повідомленнями, Telegram Bot API для взаємодії із заявниками та Docker для контейнеризації компонентів.

Сформовано загальну архітектуру системи, до складу якої входять API Gateway, бот-сервіс, сервіси заявок, користувачів, маршрутизації, сповіщень та аналітики. Розподіл функціональності за окремими сервісами забезпечує слабку зв’язаність компонентів, можливість їх незалежного розгортання, супроводу та масштабування.

Спроєктовано бізнес-процес обробки аварійної заявки від моменту створення звернення через месенджер до виконання та закриття. Передбачено перевірку вхідних даних, класифікацію, визначення пріоритету, автоматичне або ручне призначення відповідальної служби, контроль строків, зміну статусів і надсилання повідомлень заявнику.

Розроблено реляційну структуру бази даних, яка охоплює користувачів, заявки, типи аварій, статуси, служби, повідомлення, вкладення, сповіщення та історію змін. Центральною сутністю визначено заявку, а збереження історії її опрацювання забезпечує аудит дій і контроль життєвого циклу звернення.

Для взаємодії між компонентами реалізовано REST API та асинхронний обмін подіями через RabbitMQ. Синхронні запити використовуються для створення, перегляду й оновлення заявок, а події RequestCreated, RequestAssigned, RequestStatusChanged і RequestClosed — для маршрутизації, сповіщень та аналітики. Підтвердження доставки, повторні спроби обробки й dead-letter queue підвищують надійність міжсервісної взаємодії.

Реалізовано основні мікросервіси, вебінтерфейс оператора й адміністратора та Telegram-бот. Оператор може переглядати й фільтрувати заявки, призначати виконавців, змінювати статуси та додавати коментарі. Адміністратор керує користувачами, службами й довідниками. Telegram-бот забезпечує покрокове створення звернення, приймання тексту, фотографій і геолокації, а також інформування користувача про зміну стану заявки.

Окремо передбачено засоби безпеки та надійності: JWT-автентифікацію, рольове розмежування доступу, HTTPS, валідацію даних, захищене зберігання секретів, журналювання, моніторинг і резервне копіювання. Компоненти системи розгортаються в окремих Docker-контейнерах, а постійні томи забезпечують збереження даних після їх перезапуску.

Тестування охоплювало бізнес-логіку, REST API, роботу з PostgreSQL, авторизацію, Telegram-бот та обмін повідомленнями через RabbitMQ. Практична апробація підтвердила коректне виконання повного сценарію: створення заявки, її збереження, маршрутизацію, опрацювання оператором, зміну статусів і надсилання повідомлень заявнику.

Отже, у розділі сформовано та реалізовано цілісне програмне рішення, яке підтримує повний цикл обробки аварійних заявок, інтеграцію з Telegram, рольовий вебінтерфейс, асинхронну взаємодію та контейнерне розгортання. Отримані результати підтверджують відповідність системи визначеним вимогам і створюють основу для її подальшого розвитку та адаптації до потреб конкретної аварійної служби.

ВИСНОВКИ

У кваліфікаційній роботі розв’язано задачу проєктування та розробки системи обробки заявок для аварійних служб на основі мікросервісної архітектури та месенджерів. Актуальність роботи зумовлена необхідністю підвищення оперативності прийому звернень, зменшення навантаження на операторів, контролю виконання заявок і своєчасного інформування користувачів.

У результаті аналізу предметної області визначено основні етапи життєвого циклу аварійної заявки: прийом, реєстрацію, класифікацію, встановлення пріоритету, маршрутизацію, виконання та закриття. Встановлено, що традиційні способи роботи зі зверненнями мають такі недоліки, як перевантаження телефонних ліній, неточність фіксації даних, дублювання заявок і недостатній зворотний зв’язок.

Огляд helpdesk-, ITSM-, CRM-, муніципальних та омніканальних систем показав, що існуючі рішення підтримують базові функції обробки звернень, однак не завжди враховують специфіку аварійних служб. Це обґрунтувало доцільність створення спеціалізованої системи з підтримкою геолокації, фотографій, автоматичної маршрутизації та взаємодії через месенджер.

Під час аналізу архітектурних підходів розглянуто монолітну, клієнт-серверну, сервісно-орієнтовану та мікросервісну архітектури. Для реалізації обрано мікросервісний підхід, який забезпечує функціональний поділ системи, незалежне масштабування компонентів, інтеграцію із зовнішніми каналами та стійкість до часткових відмов.

У процесі проєктування сформовано архітектуру, до складу якої входять API Gateway, бот-сервіс, сервіси заявок, користувачів, маршрутизації, сповіщень та аналітики, а також PostgreSQL і RabbitMQ. Спроєктовано бізнес-процес обробки заявки, структуру бази даних, REST API, правила переходів між статусами та набір асинхронних подій.

Для реалізації використано ASP.NET Core, Entity Framework Core, PostgreSQL, RabbitMQ, Telegram Bot API, Docker і Docker Compose. Обраний

стек забезпечив модульність, зручність супроводу, повторюваність розгортання та можливість подальшого розширення системи.

Реалізовано основні мікросервіси, вебінтерфейс оператора й адміністратора та Telegram-бот. Оператор може переглядати й фільтрувати заявки, призначати виконавців, змінювати статуси та контролювати історію опрацювання. Адміністратор керує користувачами, службами й довідниками. Telegram-бот забезпечує покрокове створення заявки, приймання тексту, фотографій і геолокації та надсилання статусних повідомлень.

Для асинхронної взаємодії реалізовано події RequestCreated, RequestAssigned, RequestStatusChanged і RequestClosed. Використання RabbitMQ зменшило залежність між компонентами та забезпечило підтвердження доставки, повторну обробку й ізоляцію помилкових повідомлень.

У системі передбачено JWT-автентифікацію, рольове розмежування доступу, HTTPS, валідацію даних, захищене зберігання секретів, журналювання, моніторинг і резервне копіювання. Контейнерне розгортання забезпечило ізоляцію компонентів і збереження даних у постійних томах.

Модульні, інтеграційні та функціональні тести підтвердили коректність бізнес-логіки, REST API, роботи з базою даних, рольових обмежень, Telegram-бота й асинхронного обміну повідомленнями. Практична апробація засвідчила працездатність повного сценарію: від створення заявки через Telegram до її маршрутизації, опрацювання оператором, зміни статусів і сповіщення заявника.

Таким чином, мету кваліфікаційної роботи досягнуто. Створено працездатний прототип системи, що підтримує повний цикл обробки аварійних заявок, рольовий вебінтерфейс, інтеграцію з Telegram, асинхронну взаємодію та контейнерне розгортання. Подальший розвиток може передбачати підключення інших каналів зв’язку, інтеграцію з геоінформаційними платформами, розширення аналітики та автоматичне виявлення дублікатів звернень.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Про систему екстреної допомоги населенню за єдиним телефонним номером 112 : Закон України від 13.03.2012 № 4499-VI. База даних «Законодавство України» / Верховна Рада України.
2. Telegram Bot API : офіційна документація Telegram для розробників. Telegram, 2026.
3. Microservices Architecture Style. Azure Architecture Center : офіційна документація Microsoft. Microsoft, 2025
4. Incident Management Process. ServiceNow Product Documentation. ServiceNow, 2026.
5. Azure Well-Architected Framework. Reliability. Microsoft Learn. Microsoft, 2026.
6. Ticketing System and Help Desk Software. Zendesk : офіційний вебсайт. Zendesk, 2025.
7. Exploring Incident Management. ServiceNow Product Documentation. ServiceNow, 2026.
8. What Are Queues? Jira Service Management Cloud Documentation. Atlassian, 2026.
9. Create and Manage Basic Queues for Cases. Dynamics 365 Customer Service Documentation. Microsoft, 2025.
10. Overview of ASP.NET Core. Microsoft Learn. Microsoft, 2025.
11. Overview of Entity Framework Core. Microsoft Learn. Microsoft, 2024.
12. PostgreSQL Documentation. PostgreSQL Global Development Group, 2026.
13. Overview of YARP. Microsoft Learn. Microsoft, 2025.
14. RabbitMQ Documentation and Tutorials. Broadcom, 2026.
15. Docker Compose Documentation. Docker Inc., 2026.
16. OpenAPI Specification. OpenAPI Initiative, 2026.
17. Data Considerations for Microservices. Azure Architecture Center. Microsoft, 2025.

18. Мартиненко Г. Ю. Концептуальне та логічне проєктування реляційних баз даних : навчальний посібник. Харків : НТУ «ХПІ», 2023.
19. Павловський В. І., Петрашенко А. В., Победа Д. В. Бази даних та засоби управління : підручник. Київ : КПІ ім. Ігоря Сікорського, 2024.
20. Overview of ASP.NET Core Authentication; Introduction to Authorization in ASP.NET Core. Microsoft Learn, 2026.
21. OWASP API Security Top 10 – 2023. Open Worldwide Application Security Project, 2023.
22. PostgreSQL Documentation. Backup and Restore. PostgreSQL Global Development Group, 2026.
23. The NIST Cybersecurity Framework 2.0. National Institute of Standards and Technology, 2024.
24. Overview of ASP.NET Core MVC. Microsoft Learn. Microsoft, 2026.
25. Role-based Authorization in ASP.NET Core MVC. Microsoft Learn. Microsoft, 2026.
26. Data Tables: Four Major User Tasks. Nielsen Norman Group, 2022.
27. Web Content Accessibility Guidelines (WCAG) 2.2. W3C Recommendation. World Wide Web Consortium, 2024.
28. Telegram Bot Features and Webhooks. Telegram Documentation. Telegram, 2026.
29. Implementing Event-Based Communication Between Microservices. .NET Microservices Architecture. Microsoft Learn, 2025.
30. Exchanges and Message Routing. RabbitMQ Documentation. Broadcom, 2026.
31. Integration Tests in ASP.NET Core. Microsoft Learn. Microsoft, 2026.
32. Getting Started with xUnit.net for .NET. xUnit.net Documentation, 2026.
33. Testcontainers for .NET Documentation. Testcontainers, 2026.
34. ISO/IEC 25010:2023. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model. International Organization for Standardization, 2023.

35. ISO/IEC 25023:2016. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Measurement of system and software product quality. International Organization for Standardization, 2016.
36. Use Docker Compose in Production. Docker Documentation. Docker Inc., 2026.
37. Docker Engine Storage: Volumes. Docker Documentation. Docker Inc., 2026.
38. ASP.NET Core Runtime Environments. Microsoft Learn. Microsoft, 2025.
39. Рабенюк Д. А. Система обробки заявок для аварійних служб на основі мікросервісної архітектури та месенджерів : репозиторій програмного проєкту. GitHub, 2026. URL: https://github.com/danil-rabeniuk/emergency-request-processing-system (дата звернення: 25.05.2026).
40. Моркун Н. В., Завсегдашня І. В. Методичні вказівки до виконання кваліфікаційної роботи бакалавру для студентів спеціальності 122 “Комп’ютерні науки”. Кривий Ріг : Видавничий центр КНУ, 2021. 50 с.
41. ДСТУ 3008:2015. Інформація та документація. Звіти у сфері науки і техніки. Структура і правила оформлення. Київ, ДП «УкрННЦ», 2015. 26с. (Інформація та документація).
42. ДСТУ 8302:2015. Бібліографічне посилання. Загальні вимоги та правила складання Київ, ДП «УкрННЦ», 2016. 16 с. (Інформація та документація).