Київський столичний університет імені Бориса Грінченка
Факультет інформаційних технологій та математики
Кафедра комп’ютерних наук

Допущено до захисту
Завідувач кафедри комп’ютерних наук,
доктор технічних наук, професор

Андрій БОНДАРЧУК
«01» червня 2026 р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня «бакалавр»

Спеціальність 122 Комп’ютерні науки

Освітня програма 122.00.01 Інформатика

Тема: МОДЕЛЮВАННЯ АРХІТЕКТУРИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ НА ОСНОВІ МІКРОСЕРВІСНОГО ТА СЕРВЕРЛЕСНОГО ПІДХОДІВ

Виконав
Студент групи ІНб-1-22-4.0д
Принделас Валентин Леонідович

_____

підпис)

Науковий керівник

Доцент кафедри комп`ютерних наук, к.т.н., доцент
Абрамов В.О

Київ – 2026

Київський столичний університет імені Бориса Грінченка
Факультет інформаційних технологій та математики
Кафедра комп'ютерних наук

«Затверджую»
Завідувач кафедри комп’ютерних наук,
доктор технічних наук, професор
Андрій БОНДАРЧУК
31 жовтня 2025 року

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ БАКАЛАВРА

«Моделювання архітектури програмного забезпечення на основі мікросервісного та серверлесного підходів»

Виконавець – студент групи ІНб-1-22-4.0д,
спеціальності 122 Комп’ютерні науки,
освітньої програми 122.00.01 Інформатика
Принделас Валентин Леонідович

1.Вихідні дані: законодавство України у сфері ІТ; наукові публікації з програмної інженерії; офіційна документація Node.js, MongoDB, Docker, RESTAPI, хмарних і serverless-технологій; матеріали з мікросервісної архітектури та розподілених систем.

2. Основні завдання: проаналізувати сучасні підходи; дослідити мікросервісну та serverless-архітектури; сформувати вимоги; обґрунтувати стек і архітектуру; розробити та реалізувати систему; провести тестування; оформити пояснювальну записку.

3. Пояснювальна записка: Обсяг – до 45 стор. формату А4 комп’ютерного набору з дотриманням вимог стандарту і методичних рекомендацій кафедри.

4. Графічні матеріали: не передбачено.

5. Додатки: лістинги програмного коду, конфігураційні файли, приклади API-запитів, фрагменти бази даних та результати тестування.

6. Строк подання роботи на кафедру: 1 червня 2026 року

Науковий керівник
Доцент кафедри комп`ютерних наук
_____ Абрамов В.О.
31 жовтня 2025 року

Виконавець:
_____Принделас В.Л.
31 жовтня 2025 року

КАЛЕНДАРНИЙ ПЛАН

№	Етап виконання роботи	Термін виконання	Примітка
1	Затвердження теми кваліфікаційної роботи бакалавра	31.10.25	Виконано
2	Аналіз проблеми, вивчення аналогів, формування цілей і завдань	01.11.25 – 11.01.25	Виконано
3	Дослідження теоретичних основ мікросервісної та serverless-архітектур	26.01.26 – 15.03.26	Виконано
4	Написання першого розділу кваліфікаційної роботи	16.03.26 – 31.03.26	Виконано
5	Аналіз вимог до системи та проєктування архітектурної моделі цифрового сервісу	01.04.26 – 15.04.26	Виконано
6	Обґрунтування технологічного стеку, структури системи та моделі розгортання	16.04.26 – 30.04.26	Виконано
7	Розробка програмного рішення на основі Node.js, Express, MongoDB та Mongoose	01.05.26 – 15.05.26	Виконано
8	Підготовка пояснювальної записки, графічних матеріалів, додатків	25.05.26 – 12.06.26	Виконано
9	Захист кваліфікаційної роботи	17.06.26	Виконано

Науковий керівник
Доцент кафедри комп`ютерних наук
Абрамов В.О._____

Виконавець:

Принделас В.Л.

РЕФЕРАТ

Кваліфікаційна робота: 99 с., 13 рис., 26 табл., 53 посилання.

Актуальність дослідження зумовлена потребою цифровізації процесів продажу автомобілів, оскільки паперовий облік, окремі таблиці та застарілі монолітні системи вже не забезпечують достатнього контролю, масштабованості й ефективності. Тому важливим є впровадження сучасних архітектурних рішень, зокрема мікросервісної та serverless-архітектур, які підвищують гнучкість, надійність і економічність інформаційних систем.

Об’єкт дослідження є бізнес-процеси обліку та управління продажем автомобілів у цифровому середовищі.

Предмет дослідження - архітектурні підходи до побудови інформаційних систем, зокрема мікросервісна та serverless-архітектури, як основа цифрового сервісу автоматизації обліку продажу автомобілів.

Метою роботи є розробка та обґрунтування архітектурної моделі цифрового сервісу автоматизації обліку продажу автомобілів із використанням мікросервісного та serverless-підходів для забезпечення масштабованості, ефективності та надійності системи.

Завдання:

1. проаналізувати сучасний стан автоматизації процесів продажу на прикладі продажу автомобілів та виявити недоліки існуючих рішень;
2. дослідити принципи та особливості мікросервісної і serverless-архітектур;
3. сформувати вимоги до інформаційної системи обліку продажу автомобілів;
4. розробити архітектурну модель цифрового сервісу з розподілом функціональних компонентів;
5. реалізувати програмне рішення на основі обраного технологічного стеку;

6. провести тестування та оцінити ефективність розробленої системи.

Основні результати. У результаті виконання роботи було систематизовано теоретичні основи сучасних архітектурних підходів до розробки інформаційних систем; розроблено концепцію цифрового сервісу автоматизації обліку продажу автомобілів; спроєктовано архітектуру системи з використанням мікросервісного підходу для основної бізнес-логіки та serverless-компонентів для подієвих процесів; реалізовано ключові функціональні модулі: управління каталогом автомобілів, клієнтська підсистема, обробка договорів, фінансовий облік і звітність; обґрунтовано доцільність комбінованого використання архітектур для оптимізації продуктивності та витрат; проведено тестування системи та підтверджено її відповідність функціональним і нефункціональним вимогам.

Практичне значення дослідження полягає у можливості впровадження розробленого програмного рішення в діяльність автосалонів і дилерських центрів для автоматизації процесів обліку продажу, підвищення прозорості фінансових операцій, зменшення кількості помилок і оптимізації управлінських рішень.

Ключові слова: мікросервісна архітектура, serverless, цифровий сервіс, облік продажу, автомобілі, інформаційна система, Node.js, MongoDB, автоматизація, розподілені системи

СПИСОК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

Мікросервіс	Автономний програмний компонент, який реалізує окрему бізнес-функцію та взаємодіє з іншими сервісами через визначені інтерфейси.
API	Application Programming Interface; прикладний програмний інтерфейс для взаємодії між програмними компонентами.
AWS	Amazon Web Services; хмарна платформа для розгортання, зберігання даних і виконання серверних або serverless-компонентів.
Docker	Платформа контейнеризації, що дозволяє пакувати застосунок разом із залежностями та запускати його в ізольованому середовищі.
MongoDB	Документоорієнтована NoSQL база даних, що зберігає дані у вигляді документів.
Node.js	Серверне середовище виконання JavaScript-коду.
NoSQL	Клас нереляційних баз даних, що не обмежуються традиційною табличною моделлю SQL.
npm	Node Package Manager; менеджер пакетів для середовища Node.js.
ODM	Object Data Modeling; підхід до моделювання об’єктних даних для документоорієнтованих баз даних.
OpenAPI	Специфікація для опису REST API, на основі якої може створюватися документація та клієнтські інтеграції.
PostgreSQL	Реляційна система управління базами даних із підтримкою транзакцій і SQL.
Prometheus	Система збору та зберігання метрик для моніторингу програмних сервісів.
Puppeteer	Node.js-бібліотека для автоматизованої роботи з браузером Chromium, зокрема для генерації PDF.
RabbitMQ	Брокер повідомлень для організації асинхронної взаємодії між сервісами.
RBAC	Role-Based Access Control; рольова модель керування доступом до ресурсів системи.
Redis	Швидке сховище даних у пам’яті, яке може використовуватися для кешування та допоміжних операцій.
REST	Representational State Transfer; архітектурний стиль побудови вебсервісів.
Serverless	Архітектурний підхід, за якого розробник не керує серверною інфраструктурою напряму, а код виконується через події або керовані сервіси.

ЗМІСТ

ВСТУП

У сучасних умовах цифровізації бізнес-процесів інформаційні системи стають невід'ємною складовою ефективного управління підприємствами. Автомобільний ринок характеризується високою конкуренцією, значними фінансовими оборотами та великою кількістю супровідних операцій, що пов'язані з продажем транспортних засобів. У таких умовах використання паперових журналів, розрізнених електронних таблиць або частково автоматизованих рішень не забезпечує належного рівня контролю, прозорості та оперативності обробки інформації[4, 14, 30].

Процес продажу автомобіля включає низку взаємопов'язаних етапів: ведення каталогу транспортних засобів, управління статусами (у наявності, резерв, продано), роботу з клієнтами, оформлення договорів купівлі-продажу, проведення фінансових розрахунків, нарахування комісій, формування звітності та зберігання історії операцій. Кожен із зазначених процесів пов'язаний із обробкою структурованих і неструктурованих даних, які повинні зберігатися коректно та бути доступними в режимі реального часу. За відсутності централізованої системи управління зростає ризик помилок, дублювання даних та втрати інформації.

Актуальність теми зумовлена необхідністю створення сучасного цифрового сервісу, який забезпечує автоматизацію обліку продажу автомобілів із використанням ефективних архітектурних підходів[4, 24, 28]. Такий сервіс повинен відповідати вимогам масштабованості, надійності, безпеки та економічної доцільності. Водночас він має бути адаптивним до змін ринкових умов і вимог законодавства.

Мета дослідження – розробка та обґрунтування архітектурної моделі цифрового сервісу автоматизації обліку продажу автомобілів на основі мікросервісної та serverless-архітектур, що забезпечить підвищення ефективності управління, прозорість операцій та масштабованість системи.

Для досягнення поставленої мети необхідно вирішити такі завдання:

1. Проаналізувати сучасний стан автоматизації бізнес-процесів продажу автомобілів, виявити недоліки традиційних підходів (паперові журнали, розрізнені електронні таблиці, монолітні системи) та визначити вимоги до нової інформаційної системи.

2. Дослідити теоретичні основи мікросервісної та serverless-архітектур, їх ключові принципи, переваги, недоліки та особливості застосування в умовах висококонкурентного автомобільного ринку.

3. Розробити архітектурну модель цифрового сервісу, що включає розподіл функціональних компонентів (каталог транспортних засобів, управління статусами, робота з клієнтами, договірна та фінансова підсистеми, звітність), схему взаємодії мікросервісів та механізми serverless-виконання подій.

4. Обґрунтувати ефективність запропонованої архітектурної моделі з точки зору масштабованості, надійності, безпеки, економічної доцільності та адаптивності до змін законодавства й ринкових умов, а також сформувати рекомендації щодо її практичного впровадження.

Об’єкт дослідження – бізнес-процеси обліку та управління продажем автомобілів на підприємствах автомобільного ринку в умовах цифровізації.

Предмет дослідження – мікросервісна та serverless-архітектури як основа побудови сучасного цифрового сервісу автоматизації обліку продажу автомобілів.

Методи дослідження:

1. Аналіз стану бізнес-процесів. Критичний аналіз недоліків традиційних підходів. Синтез вимог до нової інформаційної системи.

2. Аналіз мікросервісної та serverless-архітектур, їх ключові принципи, переваги, недоліки.

3. Розробка архітектурної моделі цифрового сервісу.

4. Тестування, аналіз і обґрунтування ефективності запропонованих рішень.

Практичне значення роботи у тому, що дослідження теоретичних основ мікросервісної та serverless-архітектур і їх застосування для побудови цифрового сервісу автоматизації обліку продажу автомобілів є своєчасним і практично значущим завданням. Розробка та обґрунтування відповідної архітектурної моделі сприятиме підвищенню ефективності управління, зменшенню кількості помилок у фінансових операціях та забезпеченню стабільного функціонування інформаційної системи в умовах зростаючого навантаження.

Галузь застосування системи це автоматизація в умовах розширення мереж автосалонів, збільшення обсягів продажів та інтеграція з фінансовими установами, сервісними центрами та електронними сервісами документообігу.

РОЗДІЛ 1
ТЕОРЕТИЧНІ ОСНОВИ МІКРОСЕРВІСНОЇ ТА SERVERLESS АРХІТЕКТУР

1.1 Поняття та роль архітектури програмного забезпечення

Архітектура програмного забезпечення є фундаментальною характеристикою будь-якої інформаційної системи та визначає її загальну організацію, структуру та принципи функціонування. Під архітектурою розуміють сукупність базових рішень щодо побудови програмної системи, які визначають її складові елементи, взаємозв'язки між ними, способи обробки даних, механізми забезпечення безпеки, а також підходи до розгортання й експлуатації. Архітектурні рішення приймаються на ранніх етапах проєктування та мають довгостроковий вплив на якість, стабільність і розвиток програмного продукту[16].

У науковій літературі архітектура програмного забезпечення трактується як високорівневе представлення системи, що відображає її ключові структурні елементи та правила їх взаємодії. На відміну від детального проєктування, яке зосереджується на окремих модулях, класах і алгоритмах, архітектура визначає глобальну модель побудови системи. Вона формує рамки, у межах яких здійснюється розробка, інтеграція компонентів та їх подальша підтримка[17].

Архітектура виконує системоутворюючу функцію. Вона визначає, як логічно розділяється функціональність, як організовується доступ до даних, яким чином відбувається обмін інформацією між підсистемами, а також як забезпечується узгодженість і цілісність даних. У сучасних інформаційних системах архітектура також охоплює питання масштабованості, відмовостійкості, безпеки та спостережуваності.

У процесі розробки цифрового сервісу автоматизації обліку продажу автомобілів архітектура відіграє ключову роль, оскільки система повинна забезпечувати коректну обробку значного обсягу різнорідної інформації. До

таких даних належать відомості про транспортні засоби, клієнтів, договори купівлі-продажу, фінансові операції, знижки, комісії, а також аналітичні звіти. Невірно обрана або недостатньо продумана архітектура може призвести до порушення узгодженості даних, ускладнення внесення змін та зниження продуктивності системи[13].

Архітектура програмного забезпечення охоплює кілька взаємопов'язаних аспектів. По-перше, це структурний аспект, який визначає поділ системи на модулі або підсистеми. По-друге, це поведінковий аспект, що описує механізми взаємодії між компонентами. По-третє, це аспект управління даними, який регламентує способи зберігання, доступу та обробки інформації. По-четверте, це інфраструктурний аспект, який включає питання розгортання, конфігурації та підтримки програмного середовища.

Структурний аспект архітектури передбачає визначення логічних меж між частинами системи. У межах цифрового сервісу автоматизації обліку продажу автомобілів доцільно виділяти окремі функціональні підсистеми, зокрема модуль управління каталогом транспортних засобів, модуль роботи з клієнтами, модуль оформлення договорів і продажів, модуль фінансових розрахунків, модуль формування звітності та модуль управління доступом користувачів. Чітке розмежування відповідальності між підсистемами зменшує взаємозалежність компонентів та полегшує подальшу модифікацію системи.

Поведінковий аспект архітектури визначає механізми взаємодії між підсистемами. У межах інформаційної системи такі механізми можуть реалізовуватися через програмні інтерфейси, внутрішні сервіси або механізми обміну повідомленнями. Важливою вимогою є забезпечення узгодженості обміну даними та коректної обробки виняткових ситуацій. У системі обліку продажу автомобілів, наприклад, операція продажу повинна синхронно змінювати статус автомобіля, створювати фінансові записи та оновлювати аналітичні показники. Архітектура повинна гарантувати, що ці дії виконуються узгоджено та не призводять до виникнення суперечливих станів.

Аспект управління даними є одним із найважливіших. У системах обліку особливого значення набуває забезпечення цілісності та несуперечливості інформації. Для цього використовуються механізми транзакційності, обмеження цілісності на рівні бази даних, а також контроль доступу до критичних операцій. Архітектурні рішення визначають, чи зберігатимуться всі дані в єдиній базі, чи буде застосовано розподілене зберігання, а також як реалізовуватиметься резервне копіювання та відновлення.

Інфраструктурний аспект архітектури передбачає вибір моделі розгортання системи, організацію серверного середовища, налаштування мережевої взаємодії та механізми оновлення програмного забезпечення. У сучасних умовах інформаційні системи можуть розгортатися як у локальних дата-центрах, так і в хмарному середовищі. Вибір відповідної моделі впливає на масштабованість, доступність і витрати на експлуатацію.

Однією з основних характеристик архітектури є її вплив на нефункціональні вимоги системи. До таких вимог належать продуктивність, надійність, безпека, масштабованість, підтримуваність і тестованість. Функціональні вимоги визначають, які операції повинна виконувати система, тоді як нефункціональні - яким чином ці операції повинні виконуватися. Саме архітектура забезпечує реалізацію нефункціональних характеристик.

Продуктивність визначається здатністю системи обробляти запити користувачів у прийнятні строки. У системі обліку продажу автомобілів це означає швидке відкриття картки транспортного засобу, оперативне формування договору та своєчасне оновлення звітів. Архітектурні рішення щодо організації бази даних, використання кешування та розподілу навантаження безпосередньо впливають на досягнення необхідного рівня продуктивності.

Надійність передбачає здатність системи зберігати працездатність у разі виникнення помилок або збоїв. Архітектура повинна передбачати механізми обробки виняткових ситуацій, журналювання подій, а також можливість

відновлення після збоїв. У контексті фінансових операцій це є критично важливим, оскільки помилки можуть мати економічні наслідки.

Безпека інформації є обов’язковою складовою архітектури. Система повинна забезпечувати автентифікацію користувачів, авторизацію доступу до ресурсів, захист персональних даних та шифрування конфіденційної інформації. Архітектурні рішення визначають, де саме здійснюється перевірка прав доступу та яким чином контролюються дії користувачів.

Масштабованість означає здатність системи адаптуватися до зростання кількості користувачів або обсягу даних. У випадку розширення мережі автосалонів або збільшення кількості продажів система повинна забезпечувати стабільну роботу без істотного зниження продуктивності. Архітектура визначає, чи можливо реалізувати горизонтальне масштабування шляхом додавання нових серверних екземплярів або вертикальне масштабування шляхом збільшення ресурсів існуючих серверів.

Підтримуваність системи пов’язана з можливістю внесення змін до програмного забезпечення без значного порушення його структури. Чітке розмежування відповідальності між модулями, стандартизовані інтерфейси взаємодії та дотримання єдиних принципів проєктування спрощують процес модифікації.

Тестованість визначає можливість перевірки коректності функціонування системи за допомогою автоматизованих або ручних тестів. Архітектура, у якій бізнес-логіка відокремлена від інтерфейсу користувача та інфраструктурних компонентів, сприяє ефективному тестуванню.

Історично розвиток архітектурних підходів відбувався від простих монолітних систем до більш складних багаторівневих і розподілених рішень. Монолітна архітектура передбачає об’єднання всіх компонентів у межах одного застосунку. Такий підхід є простим у реалізації на початкових етапах, однак зі зростанням функціональності ускладнюється його підтримка та масштабування.

У сучасних умовах значного поширення набули підходи, орієнтовані на розподіл функціональності, зокрема мікросервісна та serverless-архітектури. Незважаючи на різноманітність підходів, базові принципи архітектурного проєктування залишаються незмінними: чітке визначення меж відповідальності, мінімізація залежностей, забезпечення узгодженості даних та врахування нефункціональних вимог.

Архітектура також визначає життєвий цикл обробки основних бізнес-процесів. Наприклад, процес продажу автомобіля включає перевірку доступності транспортного засобу, створення договору, фіксацію оплати та оновлення статусу. Кожен з цих кроків повинен бути реалізований таким чином, щоб система залишалася в узгодженому стані навіть у разі переривання операції. Архітектура визначає порядок виконання дій та механізми контролю їх завершення.

1.2 Принципи мікросервісного підходу

Мікросервісна архітектура є одним із сучасних підходів до побудови складних програмних систем, що ґрунтується на розподілі функціональності на окремі автономні сервіси. Під мікросервісом розуміють самостійну програмну складову, яка реалізує чітко визначену бізнес-функцію, має власний життєвий цикл, може розгортатися незалежно від інших компонентів та взаємодіє з ними через стандартизовані інтерфейси. На відміну від монолітної архітектури, де всі функції об’єднані в межах одного застосунку, мікросервісний підхід передбачає побудову системи як сукупності невеликих, логічно завершених підсистем[16, 20].

Основна ідея мікросервісної архітектури полягає в декомпозиції системи за бізнес-доменами. Кожен сервіс відповідає за окрему предметну область та інкапсулює пов’язану з нею логіку. У контексті цифрового сервісу автоматизації обліку продажу автомобілів така декомпозиція може передбачати виділення сервісу управління каталогом транспортних засобів, сервісу роботи з клієнтами, сервісу оформлення договорів, сервісу фінансових

операцій, сервісу аналітики та звітності, а також сервісу автентифікації й управління доступом. Кожен із зазначених сервісів реалізує власні бізнес-процеси та несе відповідальність за відповідний набір даних[11, 21].

Одним із ключових принципів мікросервісного підходу є принцип єдиної відповідальності. Сервіс повинен виконувати одну логічно завершену функцію або обслуговувати один домен. Такий підхід зменшує складність кожного окремого компонента та спрощує його розуміння і супровід. У системі обліку продажу автомобілів, наприклад, сервіс фінансового обліку не повинен містити логіку управління складом транспортних засобів, оскільки це різні бізнес-процеси.

Наступним принципом є автономність сервісів. Автономність означає, що кожен мікросервіс може бути розроблений, протестований, розгорнутий і масштабований незалежно від інших. Це дозволяє вносити зміни до окремих частин системи без необхідності перевипуску всієї програмної платформи. Наприклад, якщо виникає потреба змінити алгоритм розрахунку знижок або комісій менеджерів, достатньо оновити відповідний сервіс, не впливаючи на роботу модулів управління клієнтами або аналітики.

Важливою характеристикою автономності є наявність власного сховища даних у кожного сервісу. Принцип ізоляції даних передбачає, що база даних або схема, яка обслуговує конкретний сервіс, не використовується безпосередньо іншими сервісами. Обмін інформацією здійснюється через інтерфейси прикладного програмування або механізми обміну повідомленнями. Такий підхід забезпечує слабку зв’язність між компонентами та зменшує ризик неконтрольованих залежностей.

Принцип слабкої зв’язності є фундаментальним для мікросервісної архітектури. Сервіси повинні бути максимально незалежними один від одного та взаємодіяти лише через чітко визначені контракти. Зміни у внутрішній реалізації одного сервісу не повинні вимагати змін у коді інших. Це досягається шляхом стандартизації форматів обміну даними, використання версіонування інтерфейсів та чіткого документування API.

Поряд із слабкою зв’язністю важливим є принцип високої згуртованості. Кожен сервіс повинен містити тісно пов’язані функції, які логічно належать до одного домену. Висока згуртованість сприяє кращій структуризації коду та підвищує якість програмного забезпечення. У системі обліку продажу автомобілів сервіс договорів повинен охоплювати всі операції, пов’язані зі створенням, редагуванням, збереженням і контролем стану договорів, але не містити сторонніх функцій.

Принцип незалежного масштабування є однією з основних переваг мікросервісного підходу. Оскільки кожен сервіс розгортається окремо, можливо масштабувати лише ті компоненти, які зазнають найбільшого навантаження. Наприклад, у період формування щомісячної звітності може виникнути підвищене навантаження на сервіс аналітики. У такому випадку доцільно збільшити ресурси саме для цього сервісу, не змінюючи конфігурацію інших компонентів (рис. 1.1).

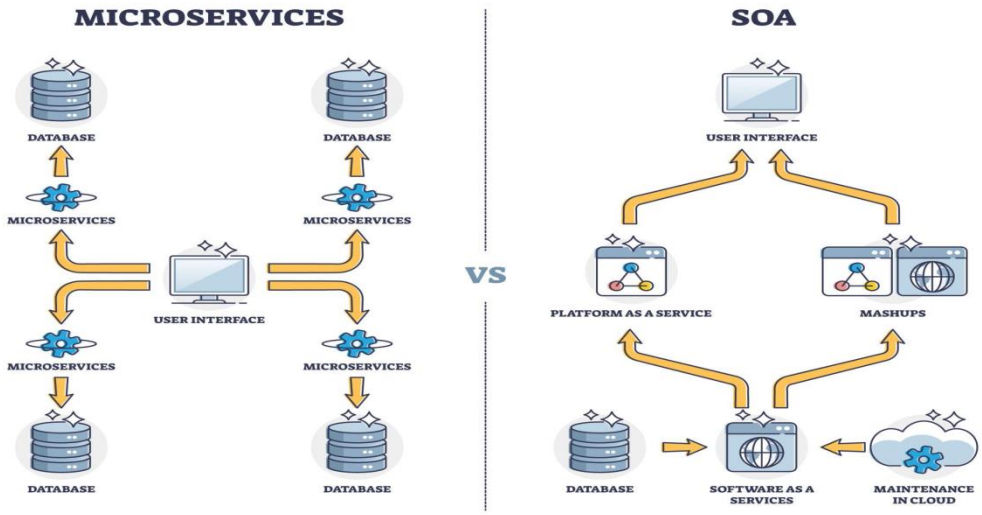

Рис. 1.1. Приклад архітектурних підходів

Ще одним важливим принципом є децентралізація управління. У мікросервісній архітектурі допускається використання різних технологій та інструментів для реалізації окремих сервісів. Це означає, що кожен сервіс може бути реалізований з урахуванням специфіки його функціональності.

Наприклад, сервіс аналітики може використовувати спеціалізовану базу даних для обробки великих обсягів інформації, тоді як сервіс транзакцій працюватиме з реляційною базою даних для забезпечення цілісності операцій.

Однак децентралізація потребує чіткої координації та стандартизації. Незважаючи на можливість використання різних технологій, необхідно дотримуватися єдиних принципів безпеки, моніторингу та логування. Відсутність централізованого контролю може призвести до зростання складності та труднощів у супроводі системи.

Принцип автоматизації процесів розгортання та тестування є невід'ємною складовою мікросервісної архітектури. Оскільки система складається з багатьох окремих сервісів, ручне управління оновленнями стає неефективним. Тому застосовуються підходи безперервної інтеграції та безперервного розгортання, які забезпечують швидке й безпечне впровадження змін.

Взаємодія між сервісами може здійснюватися синхронно або асинхронно. Синхронна взаємодія передбачає прямий виклик одного сервісу іншим із очікуванням відповіді. Асинхронна взаємодія реалізується через механізми обміну повідомленнями або подієву модель. Вибір способу взаємодії залежить від вимог до узгодженості та швидкості обробки. У системі обліку продажу автомобілів синхронна взаємодія може використовуватися для перевірки статусу транспортного засобу перед оформленням договору, тоді як асинхронна - для формування звітів або надсилання повідомлень.

Разом із перевагами мікросервісний підхід має і певні виклики[14, 6, 15]. Основним із них є зростання складності розподіленої системи. Необхідно забезпечити надійний механізм виявлення сервісів, балансування навантаження, централізоване логування та моніторинг. Крім того, виникає проблема узгодженості даних у розподіленому середовищі. Операції, що охоплюють кілька сервісів, потребують спеціальних механізмів координації.

Питання безпеки у мікросервісній архітектурі також набуває особливого значення. Кожен сервіс повинен перевіряти права доступу та забезпечувати

захист даних. Необхідно реалізувати централізовану систему автентифікації та механізми передачі токенів між сервісами.

Моніторинг і логування є критично важливими для забезпечення стабільної роботи. У розподіленій системі помилки можуть виникати на різних рівнях, тому необхідно впроваджувати централізовані системи збору журналів та аналізу метрик. Це дозволяє своєчасно виявляти проблеми та запобігати їх ескалації.

Застосування мікросервісного підходу в системі автоматизації обліку продажу автомобілів дозволяє забезпечити чітке розмежування функціональних зон, оптимізувати використання ресурсів і створити основу для подальшого розвитку цифрового сервісу. Водночас успішна реалізація такого підходу можлива лише за умови дотримання визначених принципів і врахування особливостей розподілених систем.

Основи serverless-архітектури

Serverless-архітектура є підходом до побудови програмних систем, за якого виконання коду здійснюється у вигляді окремих функцій, що запускаються у відповідь на події. Термін "serverless" не означає відсутність серверів як таких, а передбачає, що розробник не керує безпосередньо серверною інфраструктурою. Управління ресурсами, масштабування, балансування навантаження та забезпечення доступності покладається на хмарного провайдера. Такий підхід дозволяє зосередитися на реалізації бізнес-логіки, мінімізуючи витрати на адміністрування[1, 11].

Основою serverless-підходу є модель "Function as a Service" (FaaS) (рис. 1.2), відповідно до якої програмний код організовується у вигляді окремих функцій. Кожна функція виконує конкретну операцію та активується подією. Подіями можуть бути HTTP-запит, зміна даних у базі, завантаження файлу до сховища, повідомлення з черги або запланований таймер. Функції виконуються ізольовано та завершують роботу після обробки запиту.

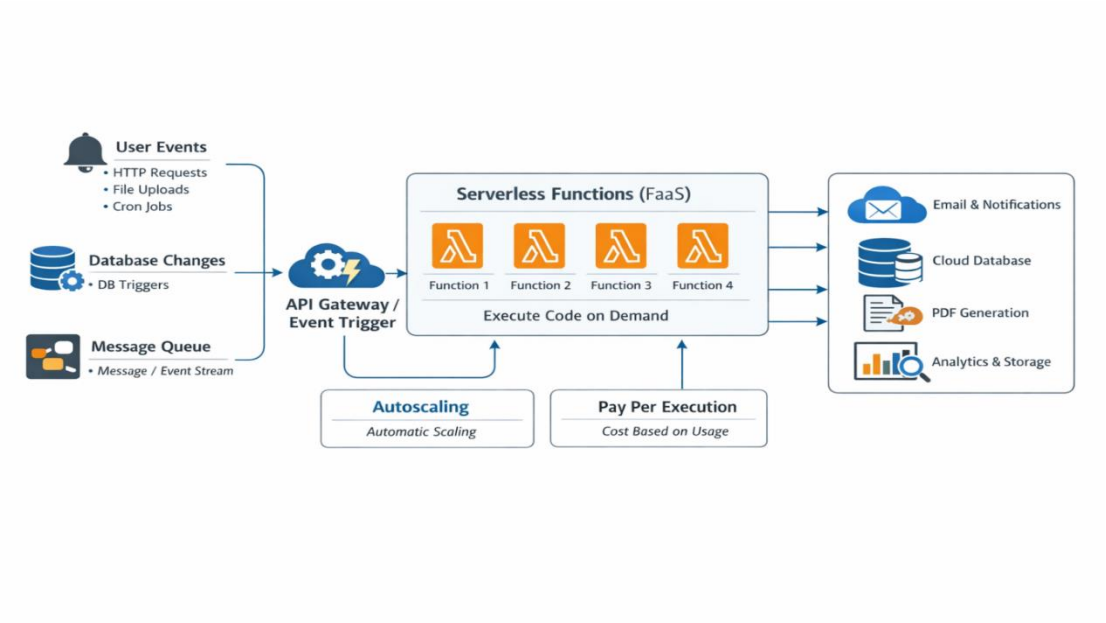

Рис. 1.2. Архітектура Function as a Service

У контексті цифрового сервісу автоматизації обліку продажу автомобілів serverless-архітектура може використовуватися для реалізації допоміжних або подієво-орієнтованих операцій. Наприклад, після створення договору система може ініціювати подію, яка запускає функцію генерації PDF-документа. Після успішної реєстрації продажу може запускатися функція надсилання електронного повідомлення клієнту або оновлення зовнішньої аналітичної системи. Такі операції не потребують постійно активного серверного процесу та можуть виконуватися асинхронно.

Ключовою характеристикою serverless-архітектури є подієва модель взаємодії. Система реагує на виникнення певних подій, а не функціонує у вигляді постійно працюючого сервера з фіксованим навантаженням. Це дозволяє більш ефективно використовувати обчислювальні ресурси, оскільки обробка виконується лише тоді, коли це необхідно.

Однією з основних переваг serverless-підходу є модель оплати за фактичне використання ресурсів. У традиційній серверній архітектурі ресурси резервуються незалежно від фактичного навантаження. У serverless-середовищі оплата здійснюється лише за час виконання функцій та обсяг використаних ресурсів. Для систем із нерівномірним навантаженням це є економічно доцільним рішенням.

Масштабування у serverless-архітектурі здійснюється автоматично. У разі збільшення кількості запитів провайдер створює необхідну кількість паралельних екземплярів функцій. Це дозволяє системі адаптуватися до змін навантаження без ручного втручання адміністратора. У цифровому сервісі обліку продажу автомобілів це може бути корисним під час пікових періодів, наприклад у момент формування звітів або масової розсилки документів.

Важливою перевагою є зменшення адміністративного навантаження. Розробнику не потрібно налаштовувати сервери, встановлювати операційну систему, контролювати оновлення безпеки або забезпечувати резервування ресурсів. Провайдер хмарної платформи відповідає за інфраструктуру, що дозволяє зосередитися на логіці прикладного рівня.

Разом із тим serverless-архітектура має певні обмеження. Одним із них є обмеження часу виконання функції. Більшість хмарних платформ встановлюють максимальну тривалість обробки одного запиту. Це ускладнює реалізацію довготривалих або ресурсоємних процесів. У такому випадку необхідно використовувати інші механізми, зокрема розбиття завдання на кілька етапів або застосування окремих сервісів обробки.

Іншим обмеженням є залежність від конкретного постачальника хмарних послуг. Serverless-функції часто реалізуються з використанням специфічних сервісів провайдера. Це може ускладнювати перенесення системи до іншого середовища. Така ситуація потребує ретельного проєктування з урахуванням мінімізації залежності від конкретних технологій.

Особливу увагу слід приділяти питанням безпеки. У serverless-середовищі функції можуть мати доступ до різних ресурсів, зокрема баз даних і сховищ. Необхідно чітко визначати права доступу кожної функції та використовувати принцип мінімальних привілеїв. Крім того, передача даних між компонентами повинна здійснюватися із застосуванням захищених протоколів.

Ще одним аспектом є моніторинг та логування. Оскільки функції виконуються короткочасно і динамічно створюються у великій кількості,

необхідно забезпечити централізований збір журналів та аналіз подій. Це дозволяє виявляти помилки та оцінювати продуктивність системи.

З погляду архітектурного проєктування serverless-підхід доцільно використовувати для реалізації ізольованих, подієвих або допоміжних операцій. У системі автоматизації обліку продажу автомобілів основні транзакційні процеси можуть залишатися в межах стабільного серверного середовища, тоді як додаткові функції, такі як генерація звітів, обробка повідомлень або інтеграція з зовнішніми сервісами, можуть бути реалізовані у serverless-моделі.

Serverless-архітектура є ефективним інструментом для побудови масштабованих і гнучких інформаційних систем[9, 27]. Вона дозволяє зменшити витрати на інфраструктуру, автоматизувати масштабування та прискорити впровадження нових функцій. Водночас її застосування потребує врахування обмежень щодо тривалості виконання, залежності від провайдера та особливостей управління безпекою і моніторингом. Раціональне поєднання serverless-компонентів із традиційними сервісами дозволяє створити збалансовану та ефективну архітектуру цифрового сервісу.

1.3 Порівняльний аналіз підходів

Мікросервісна та serverless-архітектури належать до сучасних підходів побудови розподілених інформаційних систем. Обидва підходи базуються на ідеї декомпозиції функціональності на окремі компоненти, проте реалізуються різними способами та передбачають різний рівень контролю над інфраструктурою. Їх порівняння дозволяє визначити доцільність використання кожного з підходів у межах цифрового сервісу автоматизації обліку продажу автомобілів[9, 11].

Мікросервісна архітектура передбачає створення повноцінних автономних сервісів, які працюють постійно, мають власний життєвий цикл, базу даних, механізми масштабування та моніторингу. Кожен сервіс є окремим застосунком або контейнером, який розгортається на серверній

інфраструктурі. Розробник або команда розробки відповідає за конфігурацію середовища, налаштування серверів, балансування навантаження, оновлення та контроль працездатності.

Serverless-архітектура, навпаки, орієнтована на виконання окремих функцій у відповідь на події. Код функції запускається лише тоді, коли виникає відповідний тригер, після чого завершує роботу. Інфраструктурне управління повністю покладається на хмарного провайдера. Розробник не взаємодіє безпосередньо з серверами та не відповідає за масштабування.

Основні відмінності підходів

Головна відмінність полягає в моделі виконання. Мікросервіс є постійно активним компонентом системи, тоді як serverless-функція існує лише в момент виконання. Це впливає на характер навантаження, витрати ресурсів та спосіб проєктування бізнес-процесів.

У мікросервісній архітектурі кожен сервіс може підтримувати складні транзакційні процеси, зберігати стан та взаємодіяти з іншими сервісами через API або систему обміну повідомленнями. Serverless-модель більше підходить для короткотривалих операцій без складного внутрішнього стану.

Ще однією відмінністю є рівень контролю. У мікросервісній архітектурі команда розробки має повний контроль над середовищем виконання: вибір операційної системи, налаштування контейнерів, конфігурація мережі. У serverless-підході ці аспекти визначаються постачальником хмарних послуг (табл. 1.1).

Таблиця 1.1

Порівняння архітектур

Критерій	**Мікросервісна архітектура**	**Serverless-архітектура**
Модель виконання	Постійно працюючі сервіси	Короткотривалі функції
Управління інфраструктурою	Контроль розробника	Контроль хмарного провайдера
Масштабування	Налаштовується окремо	Автоматичне
Зберігання стану	Підтримується	Обмежене

Критерій	Мікросервісна архітектура	Serverless-архітектура
Тривалість виконання	Без жорстких обмежень	Ліміти часу
Вартість	Постійні витрати	Оплата за фактичне використання
Складність адміністрування	Вища	Нижча
Гнучкість у виборі технологій	Висока	Обмежена платформою

У цифровому сервісі автоматизації обліку продажу автомобілів мікросервісний підхід доцільно застосовувати для реалізації основних бізнес-процесів. До таких процесів належать управління каталогом автомобілів, оформлення договорів, фінансовий облік та управління доступом користувачів. Ці компоненти потребують постійного доступу до даних, підтримки складних транзакцій та збереження стану. Serverless-функції можуть використовуватися для допоміжних або асинхронних операцій. Наприклад, автоматичне формування PDF-документів, надсилання повідомлень клієнтам, обробка подій після завершення продажу, синхронізація з зовнішніми системами або виконання періодичних аналітичних задач. Такий підхід дозволяє оптимізувати витрати ресурсів і спростити розгортання додаткового функціоналу (табл. 1.2).

Таблиця 1.2

Порівняння підходів

Напрям застосування	Мікросервісний підхід	Serverless-підхід	Коментар щодо доцільності
Управління каталогом автомобілів	Реалізується як окремий постійний сервіс з власною базою даних	Не рекомендовано для основної логіки	Потребує постійного доступу до даних, складних фільтрів і збереження стану
Робота з клієнтами	Окремий сервіс CRM із транзакційною підтримкою	Може використовуватися для обробки подій	Основні операції повинні бути стабільними та контрольованими

Напрям застосування	Мікросервісний підхід	Serverless-підхід	Коментар щодо доцільності
		(наприклад, сповіщення)	
Оформлення договорів	Повноцінний сервіс з логікою перевірки, статусами та аудитом	Генерація PDF може бути реалізована через функцію	Транзакційні процеси доцільно залишати у мікросервісі
Фінансовий облік	Окремий сервіс із забезпеченням цілісності даних	Може застосовуватися для періодичних розрахунків або аналітики	Вимагає збереження стану та контролю операцій
Управління доступом	Централізований сервіс авторизації та ролей	Перевірка токенів може виконуватися у функціях	Критично важливий компонент, має бути стабільним
Формування PDF-документів	Може бути частиною сервісу договорів	Доцільно реалізувати як serverless-функцію	Короткотривала подія без складного стану
Надсилання електронних повідомлень	Може бути частиною сервісу	Оптимально реалізується як подієва функція	Асинхронна операція, що не потребує постійного сервера
Синхронізація з зовнішніми системами	Через API сервісів	Подієві функції для інтеграцій	Зручно використовувати serverless для обробки зовнішніх тригерів
Аналітика та звітність	Основні агрегати формуються сервісом	Періодичні обчислення можуть виконуватися функціями	Комбінований підхід дозволяє оптимізувати навантаження

Комбінування мікросервісної та serverless-архітектур дозволяє отримати збалансовану систему [9]. Основні сервіси забезпечують стабільність, контроль і підтримку складних бізнес-операцій, тоді як serverless-компоненти забезпечують гнучкість, автоматичне масштабування та економічну

ефективність для подієвих процесів. Такий гібридний підхід також сприяє поетапному розвитку системи. На початковому етапі можуть бути реалізовані ключові мікросервіси, а з часом додаватися serverless-функції для оптимізації окремих операцій (табл. 1.2).

Таблиця 1.3

Переваги архітектури

Перевага	**Пояснення**
Баланс стабільності та гнучкості	Основні бізнес-процеси залишаються контрольованими в межах мікросервісів, а допоміжні операції реалізуються гнучко через функції
Оптимізація витрат	Serverless-компоненти оплачуються лише під час виконання
Незалежне масштабування	Мікросервіси масштабуються відповідно до навантаження, функції - автоматично
Поетапний розвиток системи	Можливість додавати функціональність без перебудови всієї архітектури
Підвищення відмовостійкості	Допоміжні процеси не впливають безпосередньо на основну логіку

Попри переваги, комбінування підходів ускладнює архітектуру. Необхідно забезпечити узгодженість даних між постійними сервісами та функціями, які виконуються за подіями. Також виникає потреба в централізованому моніторингу, журналюванні та системі управління доступом. Особливої уваги потребує проєктування інтеграцій між мікросервісами та serverless-функціями (табл. 1.4). Невірно організована взаємодія може призвести до втрати даних або некоректної обробки подій.

Таблиця 1.4

Ризики архітектури

Ризик	**Опис**
Ускладнення архітектури	Збільшується кількість компонентів та інтеграцій
Питання узгодженості даних	Необхідно контролювати взаємодію між сервісами та функціями

Ризик	Опис
Потреба в централізованому моніторингу	Логи та метрики повинні збиратися в єдиній системі
Складність інтеграції	Неправильне проєктування взаємодії може призвести до втрати або дублювання даних
Залежність від провайдера	Serverless-компоненти можуть бути прив’язані до конкретної хмарної платформи

1.4 Проблеми та виклики розподілених систем

Розподілені інформаційні системи є складними технічними структурами, у яких обробка даних здійснюється не в межах одного застосунку або одного сервера, а між кількома взаємопов’язаними компонентами. Такий підхід дозволяє підвищити масштабованість, гнучкість і продуктивність, проте водночас породжує низку специфічних проблем. На відміну від монолітної архітектури, де всі модулі працюють у межах єдиного процесу та спільної пам’яті, розподілені системи функціонують через мережеву взаємодію, що суттєво впливає на їхню поведінку[6].

Однією з базових проблем є затримки при передачі даних між сервісами. У межах одного процесу виклик функції виконується практично миттєво. У розподіленій системі кожна взаємодія між сервісами здійснюється через мережу, що додає часову затримку. У разі великої кількості запитів або складної послідовності викликів ці затримки можуть накопичуватися та впливати на загальну продуктивність системи. У цифровому сервісі обліку продажу автомобілів це може проявлятися у сповільненні оформлення договору або затримці оновлення статусу транспортного засобу.

Іншою важливою проблемою є часткові збої. У розподіленому середовищі відмова одного сервісу не обов’язково означає повне припинення роботи системи. Проте така ситуація створює складність, оскільки інші сервіси можуть очікувати відповідь від недоступного компонента. Наприклад, якщо сервіс фінансового обліку тимчасово недоступний, а сервіс оформлення

договорів продовжує працювати, може виникнути некоректний стан даних. Саме тому необхідно передбачати механізми обробки відмов.

У розподілених системах неможливо гарантувати постійну доступність усіх компонентів. Тому застосовуються механізми повторних спроб виконання операцій. Якщо запит не був оброблений з першої спроби через тимчасову помилку мережі або перевантаження сервісу, система може повторити його через певний інтервал часу. Такий підхід підвищує надійність, проте потребує обережності, щоб уникнути дублювання операцій, особливо у фінансових транзакціях.

Важливим інструментом підвищення продуктивності є кешування. Кеш дозволяє зберігати результати частих запитів і повторно використовувати їх без звернення до віддаленого сервісу. У системі обліку продажу автомобілів кешування може застосовуватися для відображення списку доступних транспортних засобів або довідників. Водночас необхідно забезпечити актуальність даних, щоб уникнути відображення застарілої інформації.

Балансування навантаження є ще одним важливим аспектом. У розподілених системах один і той самий сервіс може бути розгорнутий у кількох екземплярах. Балансувальник навантаження розподіляє запити між ними, запобігаючи перевантаженню окремих компонентів. У системі автоматизації обліку продажу автомобілів це дозволяє підтримувати стабільну роботу під час пікових періодів.

Окрему складність становить забезпечення узгодженості даних. У централізованій системі транзакції можуть виконуватися в межах однієї бази даних. У розподіленому середовищі дані можуть зберігатися в різних сервісах. Це ускладнює підтримку єдиного узгодженого стану. Наприклад, продаж автомобіля передбачає зміну його статусу, створення фінансового запису та оновлення аналітичних показників. Якщо одна з операцій виконується успішно, а інша - ні, виникає розбіжність.

Проблему узгодженості часто пов’язують із компромісом між доступністю та точністю даних. У розподілених системах застосовується

принцип, згідно з яким неможливо одночасно забезпечити повну узгодженість, постійну доступність і стійкість до мережевих збоїв. Тому архітектурні рішення повинні враховувати пріоритетність вимог. Для фінансових операцій у системі обліку продажу автомобілів перевага надається узгодженості даних.

Особливу увагу необхідно приділяти обробці транзакцій. У розподіленому середовищі складно забезпечити атомарність операцій, які охоплюють кілька сервісів. Для цього використовуються спеціальні механізми координації або компенсаційні сценарії. Наприклад, якщо під час оформлення продажу не вдалося зафіксувати оплату, система повинна повернути автомобіль у попередній статус.

Нижче наведено узагальнену таблицю основних проблем розподілених систем (табл. 1.5).

Таблиця 1.5

Проблеми розподілених систем

Проблема	**Опис**	**Потенційні наслідки**
Мережева затримка	Часова різниця між запитом і відповіддю	Сповільнення роботи системи
Часткові збої	Недоступність окремих сервісів	Порушення бізнес-процесів
Узгодженість даних	Складність синхронізації станів	Невірні фінансові записи
Перевантаження сервісів	Нерівномірний розподіл запитів	Відмова в обслуговуванні
Складність моніторингу	Розподіленість журналів	Ускладнення діагностики

Для мінімізації зазначених ризиків застосовуються різні технічні механізми[6] (табл. 1.6).

Таблиця 1.6

Механізми мінімізації ризиків

Механізм	Призначення	Ефект
Повторні спроби	Автоматичне повторення запиту	Підвищення надійності
Кешування	Зменшення кількості віддалених викликів	Підвищення продуктивності
Балансування навантаження	Розподіл запитів між екземплярами	Стабільність під навантаженням
Журналювання	Фіксація подій та помилок	Полегшення аналізу
Моніторинг метрик	Відстеження стану системи	Своєчасне реагування

У системі обліку продажу автомобілів особливо важливо забезпечити коректність фінансових операцій. Навіть незначна помилка у розподіленій взаємодії може призвести до неточностей у звітності або некоректного розрахунку комісій. Тому архітектурні рішення повинні передбачати додаткові перевірки, аудит дій користувачів та механізми відновлення після збоїв.

Ще одним викликом є забезпечення безпеки в умовах розподіленої взаємодії. Кожен сервіс має бути захищений від несанкціонованого доступу, а передача даних між компонентами повинна здійснюватися через захищені канали. Необхідно також впроваджувати централізовану систему автентифікації та контролю доступу.

Розподілені системи забезпечують гнучкість і масштабованість, проте супроводжуються низкою технічних викликів. Ефективне функціонування цифрового сервісу автоматизації обліку продажу автомобілів можливе лише за умови врахування проблем мережевої взаємодії, узгодженості даних, відмовостійкості та безпеки. Раціональне поєднання архітектурних механізмів дозволяє мінімізувати ризики та забезпечити стабільну роботу системи в умовах зростання навантаження.

Висновок до розділу 1

У першому розділі розглянуто теоретичну основу кваліфікаційної роботи: роль архітектури програмного забезпечення, принципи мікросервісного підходу, сутність serverless-моделі та особливості розподілених систем. Розділ показує, що архітектура є не додатковим описом до програми, а початковим рішенням, від якого залежить стабільність, підтримуваність, безпека й подальший розвиток системи. Для сервісу обліку продажу автомобілів це особливо важливо, оскільки система має працювати з каталогом машин, клієнтами, договорами, оплатами, статусами продажу та звітністю. Усі ці процеси пов’язані між собою, тому їх не можна реалізовувати як випадковий набір окремих функцій.

У підрозділі про архітектуру програмного забезпечення визначено її основні аспекти: структурний, поведінковий, інформаційний та інфраструктурний. Такий поділ дає змогу зрозуміти, як саме майбутня система має бути організована. Структурний аспект відповідає за поділ на компоненти, поведінковий - за взаємодію між ними, інформаційний - за зберігання й узгодженість даних, а інфраструктурний - за розгортання та експлуатацію. У предметній області продажу автомобілів це має пряме практичне значення: оформлення продажу повинно змінювати статус автомобіля, створювати фінансовий запис, оновлювати дані клієнта та впливати на звітність.

Аналіз мікросервісної архітектури дозволив визначити принципи, які можуть бути використані для побудови системи: єдина відповідальність, автономність, слабка зв’язність, висока згуртованість та незалежне масштабування. На їх основі доцільно виокремити домени каталогу автомобілів, клієнтів, договорів, фінансового обліку, звітності та керування доступом. Такий поділ спрощує підтримку системи й дає можливість змінювати окремі частини без порушення роботи всього застосунку. Наприклад, зміна правил звітності не повинна впливати на каталог, а оновлення фінансової логіки не має змінювати клієнтський інтерфейс.

Serverless-підхід у роботі обґрунтовано як доповнення до основної сервісної логіки. Його доцільно застосовувати для подієвих і допоміжних операцій: генерації PDF-документів, надсилання повідомлень, обробки файлів, синхронізації із зовнішніми сервісами або періодичної аналітики. Водночас основні транзакційні процеси, зокрема оформлення договору та фінансовий облік, мають залишатися в контрольованому серверному або мікросервісному контурі. Отже, у першому розділі обґрунтовано доцільність гібридної архітектури, де мікросервіси забезпечують стабільну бізнес-логіку, а serverless-компоненти підтримують короткі асинхронні процеси. Такий підхід є найбільш придатним для масштабованої й безпечної системи обліку продажу автомобілів.

РОЗДІЛ 2
ПРОЄКТУВАННЯ ТА МОДЕЛЮВАННЯ АРХІТЕКТУРИ СИСТЕМИ

2.1. Аналіз вимог до системи

Перед розробкою цифрового сервісу автоматизації обліку продажу автомобілів необхідно чітко визначити функціональні та нефункціональні вимоги. Це дозволяє уникнути невизначеності, забезпечити відповідність системи бізнес-процесам автосалону та створити основу для технічної архітектури, дизайну та тестування.

Система призначена для малого/середнього автосалону або дилерського центру. Вона автоматизує весь цикл продажу: від внесення автомобіля в каталог до укладення договору купівлі-продажу та фінансового обліку.

Система базується на принципах розподілу доступу:

- Клієнтський інтерфейс (доступний за маршрутом /) - публічний сайт для покупців.
- Адміністративна панель (доступна за маршрутом /admin) - закрита частина для співробітників салону.

Приклад реалізації на Node.js (Autorizz Car Dealership System) демонструє саме такий поділ, що спрощує підтримку та масштабованість.

Система повинна підтримувати повний цикл управління записами про транспортні засоби (табл. 2.1).

Операції:

- Додавання - форма з валідацією VIN та фото.
- Редагування - оновлення будь-якого поля (крім VIN після збереження).
- Видалення - м’яке видалення (soft delete) з можливістю відновлення.

Пошук і фільтрація - за маркою, моделлю, ціною, статусом, роком.

Таблиця 2.1

Атрибути сутності "Автомобіль"

Поле	Тип даних	Обов'язкове	Опис та обмеження
id	UUID	Так	Унікальний ідентифікатор
brand	String	Так	Марка (Toyota, BMW тощо)
model	String	Так	Модель
year	Integer	Так	Рік випуску (1900–2026)
vin	String	Так	VIN-код (17 символів)
price	Decimal	Так	Ціна в грн (з точністю до 2 знаків)
mileage	Integer	Ні	Пробіг (км)
fuelType	Enum	Так	Бензин / Дизель / Електро / Гібрид
transmission	Enum	Так	Механіка / Автомат / Робот
bodyType	Enum	Так	Седан / Позашляховик / Купе тощо
color	String	Так	Колір
status	Enum	Так	В наявності / Резерв / Продано
description	Text	Ні	Довільний опис
photos	Array<String>	Ні	Посилання на зображення (до 10)
createdAt	DateTime	Так	Дата додавання
updatedAt	DateTime	Так	Дата останнього редагування

Таблиця (табл. 2.2) відображає логіку переходів між статусами об'єкта продажу. Для кожного статусу визначено можливі зміни стану, ролі користувачів, які мають право виконувати ці зміни, а також автоматичні тригери, що ініціюють зміну статусу без ручного втручання. Така модель забезпечує контроль життєвого циклу об'єкта, запобігає некоректним переходам між станами та підтримує цілісність бізнес-процесу продажу.

Таблиця 2.2

Статуси та переходи

Поточний статус	Дозволені переходи	Хто може змінювати	Автоматичний тригер
В наявності	→ Резерв, → Продано	Менеджер / Адмін	-
Резерв	→ В наявності, → Продано	Менеджер / Адмін	Закінчення терміну резерву
Продано	Заборонено (тільки архів)	Адмін	Укладення договору

Система фіксує історію зміни статусу (audit log).

Таблиця відображає структуру сутності "Клієнт" (табл. 2.3) та визначає основні атрибути, необхідні для зберігання персональних, контактних і службових даних у системі. Для кожного поля зазначено тип даних, обов’язковість заповнення та його функціональне призначення. Така структура забезпечує коректне ведення клієнтської бази, підтримує цілісність даних, валідацію введеної інформації та створює основу для подальшої автоматизації процесів продажу й оформлення договорів.

Таблиця 2.3

Атрибути сутності "Клієнт"

Поле	Тип даних	Обов’язкове	Опис
id	UUID	Так	Унікальний ідентифікатор
fullName	String	Так	ПІБ
phone	String	Так	Телефон (з валідацією +380...)
email	String	Ні	E-mail
address	String	Ні	Адреса
passportData	String	Ні	Паспортні дані (для договору)
notes	Text	Ні	Додаткові нотатки
createdAt	DateTime	Так	Дата реєстрації

Операції: пошук за телефоном, історія покупок, злиття дублікатів.

Автоматичне створення договору на основі обраного авто та клієнта. Поля договору: номер, дата, клієнт, авто, ціна, умови оплати, підпис (електронний або PDF). Генерація PDF-договору з мокапом автосалону.

Таблиця відображає структуру сутності "Фінансова операція" (табл. 2.4) та містить атрибути, необхідні для обліку платіжних транзакцій у системі. Визначені поля забезпечують зберігання інформації про тип операції, суму платежу, спосіб оплати, поточний статус виконання та зв’язок із відповідним договором. Така структура підтримує контроль фінансових процесів, забезпечує прозорість розрахунків, відстеження платежів і формує основу для автоматизації бухгалтерського та управлінського обліку.

Таблиця 2.4

Атрибути сутності "Фінансова операція"

Поле	Тип даних	Опис
id	UUID	Унікальний ідентифікатор
contractId	UUID	Посилання на договір
type	Enum	Оплата / Повернення / Комісія
amount	Decimal	Сума
paymentMethod	Enum	Готівка / Карта / Кредит
status	Enum	Очікує / Оплачено / Відхилено
date	DateTime	Дата операції

Інтеграція з банківським API (за бажанням) для автоматичного підтвердження платежів.

У табл. 2.5 наведено основні типи звітів, що формуються в системі для моніторингу операційної діяльності, аналізу продажів, контролю фінансових показників і оцінювання ефективності роботи з клієнтами. Для кожного звіту визначено період формування, ключові метрики для аналізу та доступні формати експорту результатів. Така структура забезпечує підтримку управлінських рішень, підвищує прозорість бізнес-процесів і сприяє автоматизації аналітичної звітності.

Таблиця 2.5

Типи звітів

Звіт	Період	Ключові метрики	Експорт
Звіт по продажах	Місяць	Кількість угод, сума, середній чек	PDF/Excel
Інвентар (наявність авто)	Поточний	Авто за статусами та марками	Excel
Фінансовий баланс	Місяць	Надходження, повернення, залишок	PDF
Клієнтська база	Рік	Нові клієнти, повторні покупки	Excel
Аналітика конверсії	Тиждень	Перегляд → Бронювання → Продаж	Dashboard

На рис. 2.1 представлено діаграму варіантів використання UML (Unified Modeling Language) системи продажу автомобілів. UML є уніфікованою мовою моделювання програмних систем, а діаграма варіантів використання

UML застосовується для відображення ролей користувачів та сценаріїв їх взаємодії із системою, яка відображає взаємодію користувачів із функціональними можливостями програмного сервісу.

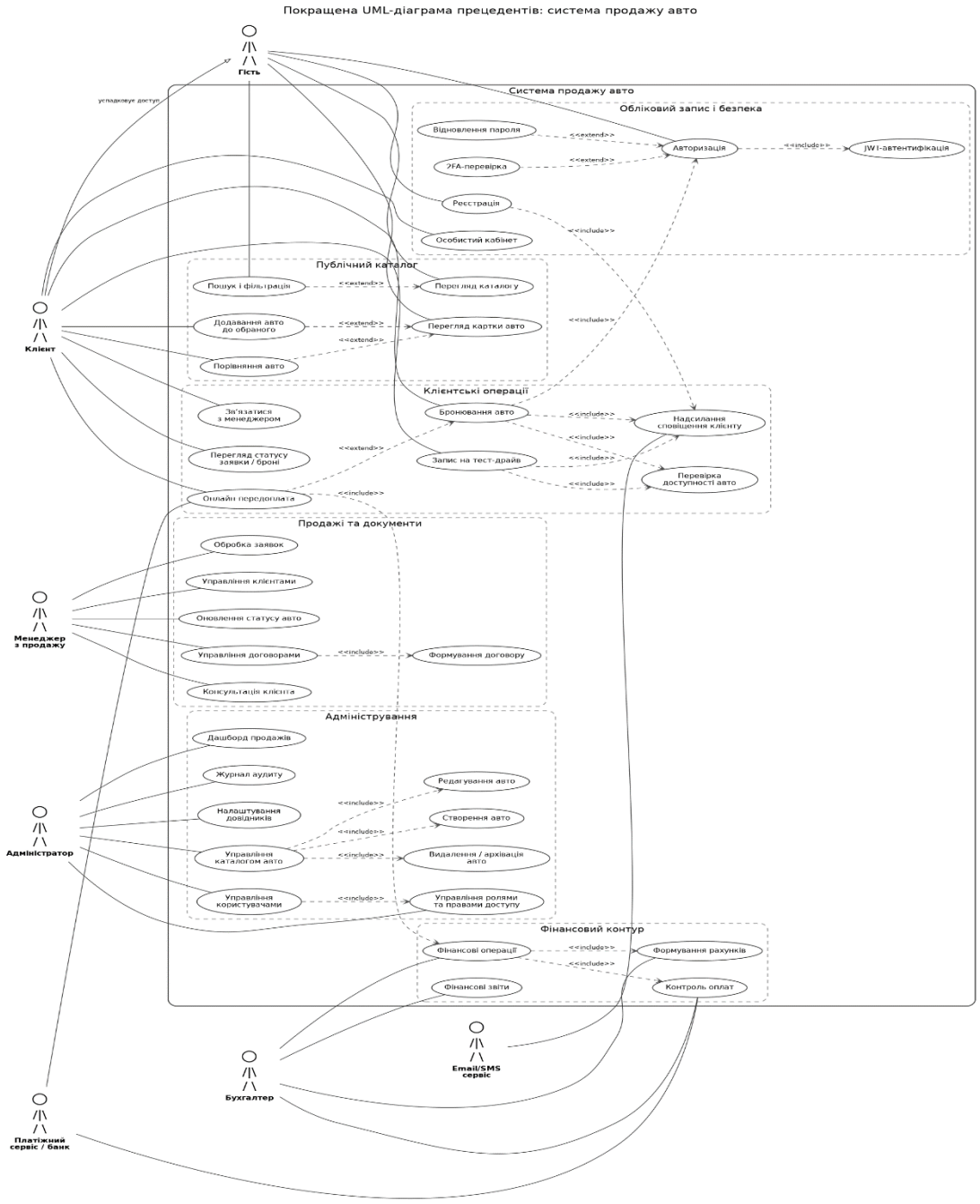

Рис. 2.1. Діаграма варіантів використання UML системи продажу автомобілів

Діаграма демонструє основних акторів системи - гостя, клієнта, менеджера з продажу, бухгалтера та адміністратора - і визначає доступні для кожної ролі сценарії взаємодії. Окремо відображено механізми автентифікації, включаючи JWT-аутентифікацію та двофакторну перевірку (2FA), а також

адміністративні операції керування користувачами, договорами, фінансами та CRUD-операції з об’єктами авто. Побудована модель формалізує функціональні межі системи, структуру ролей і бізнес-процеси, що реалізуються в межах програмного продукту.

У табл. 2.6 наведено нефункціональні вимоги до системи продажу автомобілів, що визначають критерії якості програмного продукту, його стабільності, безпеки та зручності експлуатації. Для кожної вимоги наведено відповідні метрики оцінювання та рівень пріоритетності, що дозволяєформалізувати очікувані характеристики системи на етапах проєктування, реалізації та тестування. Визначені параметри забезпечують контроль продуктивності, надійності, масштабованості, захисту даних і користувацького досвіду, формуючи основу для оцінки відповідності системи технічним і бізнес-вимогам.

Таблиця 2.6

Нефункціональні вимоги з метриками

Вимога	**Метрика / Критерій**	**Рівень пріоритету**
Продуктивність	Час відповіді ≤ 800 мс (95% запитів); підтримка 200 одночасних користувачів	Високий
Надійність	99,9% uptime; ACID-транзакції; щоденне резервне копіювання + point-in-time recovery	Критичний
Масштабованість	Горизонтальне масштабування (Docker + Kubernetes); підтримка до 10 000 авто в каталозі	Середній
Безпека	HTTPS; RBAC; захист від SQL/XSS/CSRF; GDPR-сумісність даних клієнтів	Критичний
Зручність використання	Інтуїтивний UI (Material Design / Tailwind); доступність WCAG 2.1 AA; мобільна адаптивність	Високий

Додаткові вимоги:

- Локалізація - українська + англійська.
- Логування всіх дій (audit trail).
- Підтримка темної/світлої теми.

2.2. Формування концепції архітектури

На основі проведеного аналізу предметної області, функціональних потреб майбутнього програмного продукту та особливостей обраного технологічного стеку формується загальна концепція архітектури системи. Для цифрового сервісу автоматизації обліку продажу автомобілів доцільним є використання клієнт-серверної моделі, оскільки саме вона найбільш повно відповідає вимогам до централізованого зберігання даних, багатокористувацького доступу, гнучкого адміністрування та можливості подальшого масштабування[22].

Клієнт-серверний підхід є класичним і перевіреним рішенням для побудови інформаційних систем такого типу. Його суть полягає в тому, що користувач взаємодіє з системою через клієнтський інтерфейс, тоді як основна обробка запитів, бізнес-логіка та доступ до бази даних виконуються на серверній стороні. Така модель дозволяє зосередити керування даними та ключовими процесами в одному центрі, що є особливо важливим для систем, у яких працюють різні категорії користувачів: адміністратор, менеджер з продажу, бухгалтер, а також, за потреби, клієнтська частина з публічним доступом.

У даній системі обрано стек технологій, який добре підходить для веборієнтованої реалізації та дозволяє створити гнучке, зрозуміле в підтримці програмне рішення. Серверна частина реалізується на базі Node.js, що забезпечує асинхронну обробку запитів, високу швидкодію та зручність розробки в середовищі JavaScript[53]. Використання Node.js є виправданим, оскільки ця платформа добре працює в задачах, де важлива швидка взаємодія між клієнтом і сервером, а також ефективна обробка великої кількості однотипних HTTP-запитів.

Для організації серверної логіки використовується Express.js - легкий та функціональний фреймворк для Node.js. Він відповідає за маршрутизацію, обробку запитів, роботу з middleware-компонентами та формування відповіді

клієнту. Express дозволяє чітко структурувати застосунок, розділяти модулі за функціональним призначенням і реалізовувати як публічну частину сайту, так і захищену адміністративну панель. Простота та популярність цього фреймворку також роблять його зручним для підтримки та подальшого розвитку системи.

Для зберігання даних обрано MongoDB - документоорієнтовану NoSQL базу даних. Її використання є доцільним у проєктах, де структура даних може розширюватися або змінюватися в процесі розвитку системи. У контексті цифрового сервісу продажу автомобілів це особливо зручно, оскільки картки автомобілів, профілі клієнтів, договори, статуси угод та інші сутності можуть містити різний набір атрибутів. MongoDB забезпечує гнучкість у роботі з такими даними та дозволяє зберігати їх у форматі, наближеному до JSON, що добре узгоджується з використанням JavaScript на серверній стороні.

Для взаємодії із базою даних використовується Mongoose - бібліотека моделювання об’єктних даних для MongoDB. Вона дає змогу описувати структуру документів за допомогою схем, перевіряти дані, керувати зв’язками між сутностями та виконувати запити в більш структурованому вигляді. Завдяки Mongoose код роботи з базою даних стає більш впорядкованим, зрозумілим і безпечним, а також зменшується ймовірність логічних помилок при збереженні або зміні інформації.

Для формування користувацького інтерфейсу в системі використовується Handlebars - шаблонізатор, який дозволяє генерувати HTML-сторінки на сервері. Такий підхід є доцільним для системи, де інтерфейс не потребує надскладної клієнтської логіки, але водночас має бути зручним, структурованим і динамічним. Handlebars дозволяє використовувати повторно окремі шаблонні блоки, формувати сторінки на основі даних із сервера та підтримувати єдину стилістику як для клієнтської, так і для адміністративної частини застосунку. Іншими словами, це перевірений інженерний підхід, який забезпечує стабільність, зрозумілість структури та зручність супроводу.

З погляду внутрішньої побудови система має багаторівневу архітектуру, що включає три основні рівні: рівень представлення, рівень логіки обробки та рівень доступу до даних. Така організація є принципово важливою, оскільки вона дозволяє розмежувати відповідальність між різними частинами програми й уникнути хаотичного змішування коду.

Рівень представлення (views) відповідає за відображення інформації користувачеві. Саме на цьому рівні реалізуються сторінки каталогу автомобілів, сторінки з детальним описом транспортного засобу, форми авторизації, реєстрації, бронювання, сторінки адміністративної панелі, таблиці, звіти та інші елементи інтерфейсу. Основне завдання цього рівня - забезпечити зручну й зрозумілу візуальну взаємодію користувача із системою. При цьому рівень представлення не повинен містити складної бізнес-логіки, адже його роль - саме відображення та передача введених користувачем даних до наступного рівня.

Рівень обробки логіки (routes) виконує функцію зв’язувальної ланки між інтерфейсом та даними. Саме тут обробляються HTTP-запити, перевіряються параметри, викликаються відповідні моделі, визначається порядок виконання операцій і формується відповідь користувачеві. Маршрутизація в Express.js дозволяє логічно поділити функціональність системи: окремо можуть існувати маршрути для публічного каталогу, окремо - для клієнтів, окремо - для адміністратора, менеджера чи бухгалтера. Це спрощує структуру проєкту і дозволяє контролювати доступ до конкретних функцій залежно від ролі користувача.

Рівень роботи з даними (models) забезпечує взаємодію із базою даних. Тут описуються структури основних сутностей системи: автомобілі, користувачі, клієнти, договори, фінансові записи, заявки, повідомлення та інші об’єкти. Саме моделі визначають, які поля має кожна сутність, які між ними існують зв’язки, які перевірки повинні виконуватися та як саме дані зберігаються у MongoDB. Винесення цього функціоналу в окремий рівень

дозволяє ізолювати операції з даними від інтерфейсу та бізнес-логіки, а отже - зробити код більш організованим.

Подібне розділення на рівні створює низку важливих переваг. По-перше, система стає більш зрозумілою для розробника: кожен компонент виконує свою функцію і не дублює поведінку інших частин. По-друге, спрощується супровід програмного продукту, оскільки зміни в одному рівні зазвичай не потребують повного переписування інших. Наприклад, модифікація шаблонів інтерфейсу не повинна зачіпати логіку роботи з базою даних, а зміна структури схеми документів не повинна руйнувати побудову сторінок. По-третє, така архітектура добре підходить для розширення: до системи можна додавати нові модулі, нові ролі користувачів, нові маршрути або нові сутності без повного перегляду всієї програмної реалізації.

Концепція архітектури також передбачає централізовану обробку запитів через Express.js. Усі клієнтські звернення проходять через серверний шар, де виконуються перевірка доступу, автентифікація, маршрутизація, обробка даних та формування відповіді. Такий підхід дозволяє зберігати контроль над бізнес-процесами в одному місці, а також забезпечує єдиний механізм доступу до функціональності системи. У разі використання JWT-автентифікації та, за потреби, двофакторної перевірки це ще й підвищує загальний рівень безпеки застосунку.

Окрему роль у концепції відіграє використання шаблонів для формування інтерфейсу. Завдяки Handlebars система може динамічно відображати інформацію про автомобілі, клієнтів, договори чи фінансові звіти без дублювання коду сторінок. Це особливо зручно при побудові каталогу, карток транспортних засобів, списків угод та адміністративних таблиць. Крім того, шаблонний підхід дозволяє забезпечити єдину візуальну структуру застосунку, що позитивно впливає на зручність роботи користувачів.

Не менш важливою складовою концепції є взаємодія з базою даних через Mongoose. Такий підхід дає можливість формалізувати структуру даних, підвищити надійність збереження інформації та забезпечити більш керований

доступ до колекцій MongoDB. З технічного погляду це означає, що система не працює з "сирими" даними напряму, а використовує чітко описані моделі, які містять правила валідації, логіку обробки та методи доступу до записів.

Ще одним важливим принципом архітектури є розділення користувацької та адміністративної частини. Публічний інтерфейс орієнтований на перегляд каталогу автомобілів, ознайомлення з характеристиками, використання пошуку та фільтрів, а також надсилання заявки на бронювання чи звернення до менеджера. Адміністративна панель, навпаки, є захищеним середовищем для внутрішньої роботи персоналу. У ній реалізуються функції керування товарами, клієнтами, договорами, фінансами, користувачами та аналітикою. Таке розмежування є логічним і безпечним, оскільки дозволяє чітко відділити відкритий функціонал від службових операцій.

У результаті запропонована архітектура забезпечує системі гнучкість, керованість і перспективу подальшого розвитку. Вона дозволяє не лише реалізувати базові функції цифрового сервісу автоматизації обліку продажу автомобілів, а й створює основу для розширення: підключення нових модулів, інтеграції платіжних сервісів, реалізації API, впровадження мобільного клієнта або переходу до більш масштабної мікросервісної моделі в майбутньому.

2.3. Структурна модель системи та взаємодія компонентів

Структура програмного застосунку побудована на основі архітектурного шаблону MVC (Model–View–Controller). Проте в рамках обраного стеку (Node.js + Express + MongoDB + Handlebars) (рис. 2.2) використовується спрощена інтерпретація цього підходу, характерна для серверних JavaScript-застосунків (табл. 2.7).

Класичний MVC передбачає чітке розділення на три незалежні компоненти, однак у даній реалізації роль контролера частково виконується через маршрути (routes), що є типовим рішенням для Express-застосунків.

Система організована таким чином, щоб кожен компонент відповідав за свою зону відповідальності, без зайвого “перемішування” логіки. Це не тільки робить код читабельним, а й дозволяє масштабувати систему без суттєвого ускладнення структури.

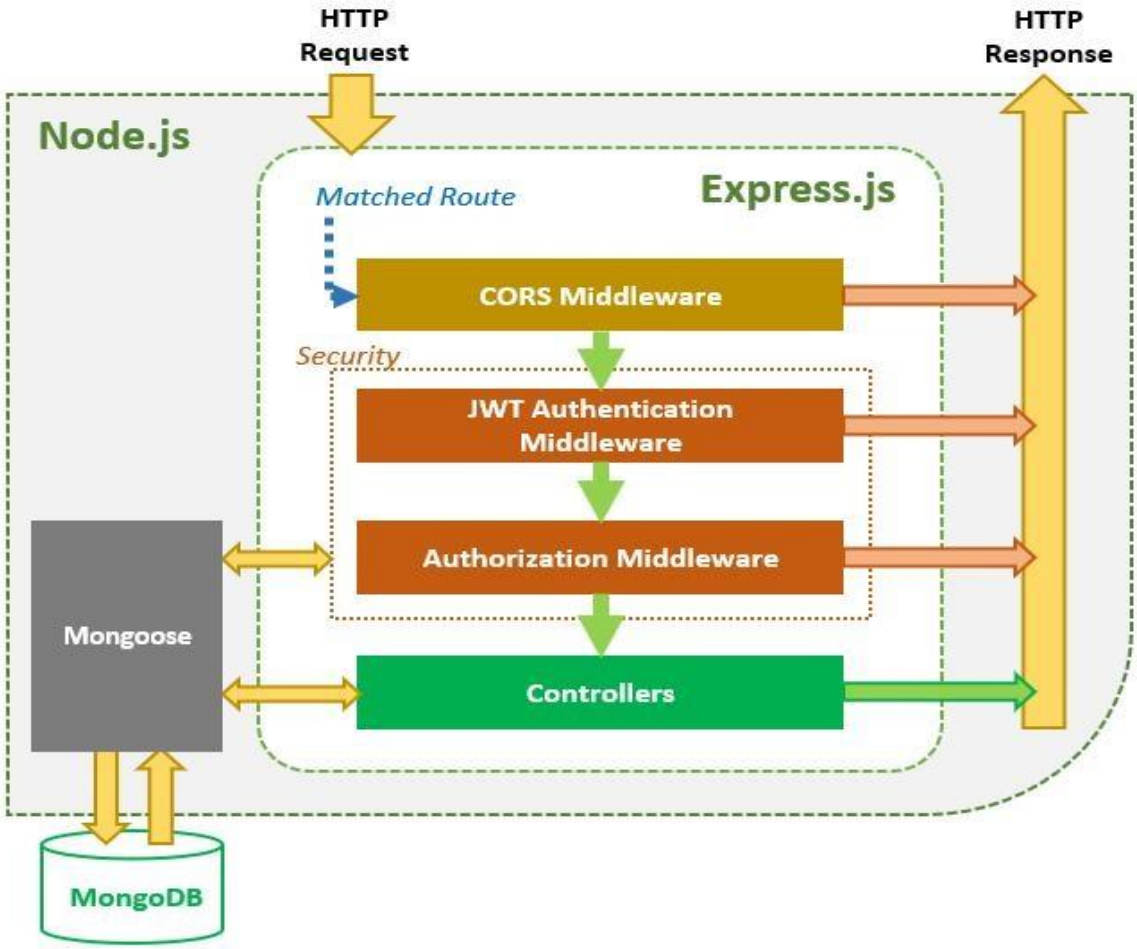

Рис. 2.2. Технологічний стек Node.js, Express, MongoDB та Handlebars

Таблиця 2.7

Класичний MVC

Компонент	Призначення
models	опис структури даних MongoDB
routes	обробка HTTP-запитів
views	відображення інтерфейсу
public	статичні ресурси
app.js	точка входу в систему

Рівень моделей є фундаментом системи - це місце, де визначається структура даних, з якою працює весь застосунок.

Моделі реалізуються за допомогою бібліотеки Mongoose, що дозволяє описувати схеми документів MongoDB у вигляді об’єктів JavaScript. Кожна модель визначає:

- поля документа (наприклад, назва авто, ціна, статус);
- типи даних;
- обмеження та валідацію;
- зв’язки між сутностями.

У системі передбачені базові моделі:

- Автомобілі - опис транспортних засобів;
- Користувачі - облікові записи з ролями;
- Замовлення - заявки або угоди.

Routes у цій архітектурі виконують роль контролерів. Саме вони приймають HTTP-запити, обробляють їх і визначають, що робити далі.

Основні задачі routes:

- прийом запиту від клієнта;
- перевірка параметрів;
- виклик моделей;
- формування відповіді;
- передача даних у views.

У системі реалізовано логічне розділення маршрутів:

- admin - адміністративна частина;
- electric - розділ електромобілів;
- gas - розділ автомобілів з ДВЗ.

Views відповідають за відображення інтерфейсу користувача. У даному випадку використовується шаблонізатор Handlebars, який дозволяє генерувати HTML на сервері.

Структура views поділяється на:

- admin - інтерфейс адміністратора;

- layouts - базові шаблони (header, footer, структура сторінки);
- користувацькі сторінки - каталог, картки авто, форми.

Головна ідея полягає в тому, що інтерфейс не містить бізнес-логіки, а лише відображає дані користувачу.

Папка public містить усі статичні файли:

- CSS (стилі);
- JavaScript (клієнтські скрипти);
- зображення.

Ці ресурси не обробляються сервером логічно - вони просто віддаються клієнту. Це дозволяє розвантажити серверну частину і пришвидшити роботу застосунку.

Файл app.js - це центр управління всім застосунком.

Його основні функції:

- ініціалізація Express-сервера;
- підключення middleware;
- налаштування шаблонізатора;
- підключення маршрутів;
- встановлення з’єднання з MongoDB;
- запуск сервера.

Взаємодія компонентів у системі відбувається за чітко визначеним сценарієм (рис. 2.3).

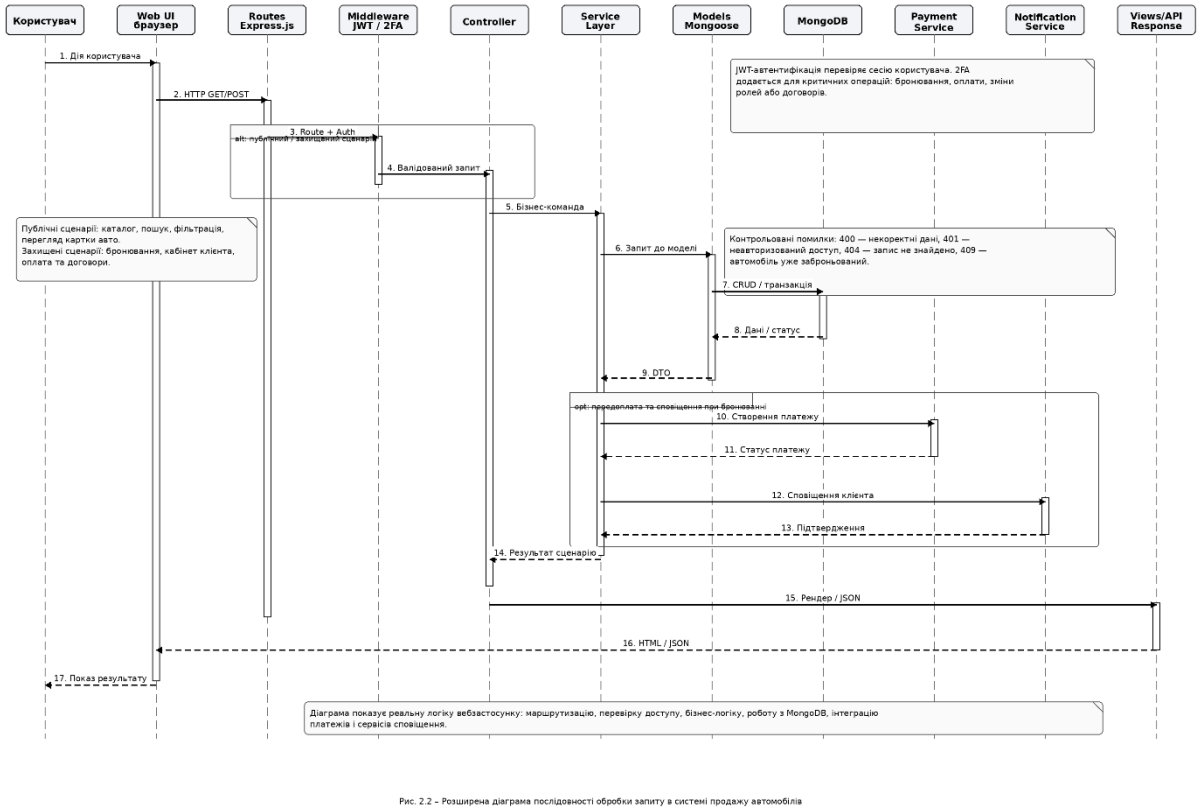

Рис. 2.2 – Розширена діаграма послідовності обробки запиту в системі продажу автомобілів

Рис. 2.3. Сценарій взаємодії компонентів системи

Процес обробки запиту в системі продажу автомобілів відбувається за таким сценарієм:

- Користувач відкриває сторінку вебзастосунку або виконує певну дію в інтерфейсі, наприклад переглядає каталог автомобілів, здійснює пошук, переходить до картки авто або створює заявку на бронювання.
- Клієнтська частина системи формує HTTP-запит і передає його на серверну частину застосунку.
- Запит надходить до рівня маршрутизації Express.js, де визначається відповідний маршрут для подальшої обробки дії користувача.
- Перед передачею запиту до основної логіки системи він проходить через middleware-рівень, де може виконуватися перевірка JWT-токена, прав доступу користувача та, за потреби, додаткова 2FA-перевірка.
- Після успішної перевірки запит передається до контролера, який визначає, яку саме операцію необхідно виконати: отримати список автомобілів, відкрити картку авто, створити бронювання, оновити статус заявки або виконати іншу дію.

- Контролер звертається до сервісного шару, де реалізується основна бізнес-логіка системи. На цьому етапі може виконуватися перевірка доступності автомобіля, підготовка даних для бронювання, обробка параметрів пошуку, перевірка коректності введених даних або формування умов запиту до бази даних.
- Сервісний шар звертається до моделей Mongoose, які забезпечують взаємодію з колекціями MongoDB.
- Моделі виконують необхідні CRUD-операції в базі даних: пошук, створення, оновлення або видалення записів, пов’язаних з автомобілями, користувачами, заявками, договорами чи фінансовими операціями.
- MongoDB повертає результат виконання запиту, після чого дані передаються назад через моделі до сервісного шару.
- Сервісний шар обробляє отримані дані та, за потреби, взаємодіє із зовнішніми сервісами. Наприклад, під час бронювання автомобіля може виконуватися звернення до платіжного сервісу для обробки передоплати.
- Якщо операція потребує інформування користувача або менеджера, система звертається до сервісу сповіщень, який надсилає повідомлення про створення заявки, зміну статусу бронювання або підтвердження фінансової операції.
- Після завершення обробки контролер формує результат виконання операції та передає його на рівень представлення або API-відповіді.
- Рівень Views/API Response формує кінцеву відповідь для користувача: HTML-сторінку, оновлені дані інтерфейсу або JSON-відповідь для клієнтської частини.
- Користувач отримує готовий результат у браузері: оновлений каталог, сторінку автомобіля, підтвердження бронювання, повідомлення про помилку або іншу відповідь відповідно до виконаної дії.

2.4. Вибір технологій та інструментів реалізації

Вибір технологічного стеку для системи автоматизації обліку продажу автомобілів обґрунтований функціональними та нефункціональними вимогами, описаними раніше: швидка обробка запитів, гнучкість даних (автомобілі з різними атрибутами, фото, історія статусів), надійність фінансових операцій, зручність адміністративної панелі та клієнтського інтерфейсу, а також швидкість розробки для малого/середнього автосалону[16, 23, 53].

Базовий стек взято з прикладу Autorizz Car Dealership System (Node.js + Express + MongoDB + Mongoose + Handlebars), але розширено сучасними практиками 2026 року для кращої масштабованості, типізації та продуктивності. Основний підхід - MEAN-подібний стек з можливістю еволюції до MERN (React замість чистого Handlebars) або NestJS для enterprise-рівня.

Переваги єдиного JavaScript/TypeScript-стеку наведено в Додатку Г:

- Одну мову для frontend і backend → швидша розробка, менша кількість помилок контексту, легше наймати розробників (в Україні та світі JS-розробників найбільше).
- Асинхронна модель Node.js ідеально підходить для I/O-операцій (багато одночасних користувачів: клієнти переглядають каталог, менеджери оформлюють угоди).
- Швидкий MVP (Minimum Viable Product) з можливістю подальшого масштабування.
- Низька вартість володіння на старті (open-source, немає ліцензій).

Альтернативи (чому не обрано):

- Python (Django/Flask) - сильний у бізнес-логіці та звітності, але повільніший на реальному часі та гірший для SPA.
- PHP (Laravel) - популярний у legacy-системах автосалонів, але менш сучасний для реального часу та мікросервісів.

- .NET / Java - надмірно важкі для середнього автосалону, вища вартість розробки.
- Go/Rust - відмінна продуктивність, але довший час розробки та менша екосистема UI.

У табл. 2.8 наведено перелік додаткових інструментів і технологічних засобів, що використовуються для реалізації окремих функціональних та нефункціональних аспектів системи продажу автомобілів. Для кожної категорії визначено відповідний інструмент і його практичне призначення в межах архітектури програмного продукту. Застосування цих засобів забезпечує валідацію даних, автоматизацію документообігу, обробку медіафайлів, логування, тестування, безперервну інтеграцію, моніторинг, підвищення безпеки та підтримку актуальної API-документації.

Таблиця 2.8

Перелік інструментів і технологічних засобів

Категорія	Інструмент	Призначення
Валідація	Joi / Zod	Валідація вхідних даних (VIN, телефон, ціна).
PDF-генерація	pdf-lib або Puppeteer	Автоматичне створення договорів купівлі-продажу.
Фотографії	Multer + Cloudinary / AWS S3	Завантаження та зберігання фото автомобілів (до 10 на авто).
Логування	Winston / Pino	Детальне логування дій (audit trail для статусів та договорів).
Тестування	Jest + Supertest / Mocha	Unit, integration та e2e-тести (особливо фінансових операцій).
CI/CD	GitHub Actions / GitLab CI	Автоматичне тестування та деплой.
Моніторинг	Prometheus + Grafana / New Relic	Продуктивність, помилки, uptime.
Безпека	Helmet, express-rate-limit, cors	Захист від атак, HTTPS, rate limiting.
Документація API	Swagger / OpenAPI	Автогенерація документації для /api.

Архітектура проекту

```
autorizz-car-system/
├── src/
│   ├── config/          # db.js, env, jwt
│   ├── controllers/     # carController, contractController, authController
```

```
│   ├── models/          # Car.js (Mongoose schema), Client.js, Contract.js
│   ├── routes/          # api/ та admin/
│   ├── middleware/      # auth, roleCheck, validation
│   ├── services/        # pdfService, emailService
│   ├── views/           # Handlebars шаблони (layouts, admin, public)
│   └── utils/           # helpers
├── public/              # static (css, js, uploaded images)
├── admin/               # окремі admin views (якв Autorizz)
├── tests/
├── docker-compose.yml
├── Dockerfile
└── package.json
```

Етапи впровадження та міграція

1. MVP (1–2 місяці): Node.js + Express + MongoDB + Mongoose + Handlebars - Каталог авто + статуси + клієнти + базові договори.

2. Розширення (2–4 місяці): Додати TypeScript, Redis, PDF, ролі, звіти.

3. Сучасний рівень: Перехід admin-панелі на React/Next.js, API-first архітектура, Docker + CI/CD.

4. Масштаб: Microservices (окремий сервіс для фінансів), Kubernetes, MongoDB Atlas для хмари.

Ризики та рекомендації

- Ризик: MongoDB не ACID за замовчуванням для складних транзакцій (фінанси + договір). Рішення: Використовувати MongoDB Transactions або гібрид з PostgreSQL для фінансової частини.
- Ризик: Handlebars стає застарілим для динамічного UI. Рішення: Планувати міграцію на Next.js з самого початку.
- Перевага: Стек залишається одним з найпопулярніших (MERN/MEAN) для швидких веб-додатків, особливо в automotive retail. Велика спільнота, легкий пошук розробників.

Цей вибір технологій забезпечує баланс між швидкістю розробки, продуктивністю, масштабованістю та зручністю підтримки. Він повністю відповідає вимогам до системи: швидка обробка запитів, надійне зберігання даних, інтуїтивний інтерфейс та безпечний адміністративний доступ.

2.5. Модель розгортання системи

Модель розгортання (Deployment Model) описує, як програмні компоненти системи автоматизації обліку продажу автомобілів розміщуються на апаратному/хмарному забезпеченні, як вони взаємодіють між собою та як забезпечується доступ користувачів[9, 53].

Система побудована на Node.js + Express + MongoDB + Handlebars (з можливістю еволюції до Next.js). Вона має чіткий розподіл:

Клієнтський інтерфейс (публічний каталог авто) - доступний за кореневим шляхом /.

Адміністративна панель - захищена частина за шляхом /admin.

Це дозволяє розділити трафік і рівні доступу: звичайні відвідувачі (потенційні покупці) працюють з публічною частиною, а співробітники автосалону (менеджери, адміністратори, бухгалтери) - з адміністративною.

Основні компоненти моделі розгортання представлені в табл. 2.9.

Таблиця 2.9

Компоненти системи та їх опис

Компонент	Опис	Технологія / Артефакт	Розташування / Node
Node.js сервер	Обробка всіх HTTP-запитів, бізнес-логіка, маршрутизація (API + views)	Node.js 20/22 + Express.js	Application Server (основний сервер)
MongoDB	Зберігання всіх даних: автомобілі, клієнти, договори, фінансові операції, користувачі	MongoDB 7/8 (або Atlas)	Database Server / Cloud Cluster
Redis (рекомендовано)	Кешування каталогу, сесій, rate limiting	Redis 7.x	Cache Server
Browser / Клієнт	Інтерфейс користувача (публічний та admin)	HTML + CSS + JS (Handlebars / React)	Кінцевий пристрій користувача
Nginx / Reverse Proxy	Обслуговування статичних файлів, SSL termination, балансування	Nginx	Web Server / Edge Node
PDF Generator	Генерація договорів купівлі-продажу	Puppeteer або pdf-lib	Вбудовано в Node.js сервер
Storage для фото	Зберігання зображень автомобілів	Cloudinary / AWS S3 / Local	External Storage

Схема взаємодії (проста трирівнева архітектура): Клієнт (Browser) → Nginx (Reverse Proxy) → Node.js Server → MongoDB / Redis

Рівні розгортання

Локальне (розробка / тестування)

- MongoDB: запуск через Docker (docker run -d -p 27017:27017 mongo) або MongoDB Compass.
- Сервер: npm install → npm run dev (nodemon).
- База: створення бази autorizz (або назва проєкту).
- Доступ:
  - Клієнтський інтерфейс: http://localhost:5000/
  - Адміністративна панель: http://localhost:5000/admin
- Переваги: швидка розробка. Недоліки: немає HA (high availability), немає масштабування.

Продакшн-розгортання

Варіант 1. Docker + Docker Compose.

Найпростіший і надійний спосіб для початку. Представлений конфігуратор-налаштовувач контейнеру Docker,

```
docker-compose.yml
version: '3.9'
services:
  mongodb:
    image: mongo:8.0
    container_name: autorizz-mongo
    restart: always
    ports:
      - "27017:27017"
    volumes:
      - mongo-data:/data/db
    environment:
      MONGO_INITDB_DATABASE: autorizz

  redis:
    image: redis:7-alpine
    container_name: autorizz-redis
    restart: always

  app:
    build: .
    container_name: autorizz-server
    restart: always
    ports:
      - "5000:5000"
    environment:
      - NODE_ENV=production
```

```
      - MONGO_URI=mongodb://mongodb:27017/autorizz
      - REDIS_URL=redis://redis:6379
      - JWT_SECRET=...
      - PORT=5000
    depends_on:
      - mongodb
      - redis

volumes:
  mongo-data:
```

Команди розгортання:

1. docker-compose up -d --build
2. Налаштування environment variables (.env не комітиться в git).
3. Nginx як reverse proxy для HTTPS.

Варіант 2. Хмарне розгортання

- MongoDB: MongoDB Atlas (managed service) - автоматичні бекапи, реплікація, scaling. Краще для більшості випадків, ніж self-hosted.
- Node.js: Render, Railway, Vercel (для API), DigitalOcean App Platform, AWS Elastic Beanstalk, Azure App Service або Fly.io.
- Переваги: немає управління серверами, автоматичне масштабування, вбудований CI/CD.

Варіант 3. Kubernetes

- Docker-образи для app, MongoDB (або Atlas), Redis.
- Deployment, Service, Ingress, Horizontal Pod Autoscaler.
- Підходить, якщо очікується >500 одночасних користувачів або кілька автосалонів.

Детальні етапи розгортання (продакшн) представлені в табл. 2.10.

Таблиця 2.10

Етапи розгортання

Етап	Дії	Інструменти / Команди	Примітки
Підготовка коду	Встановити залежності, налаштувати .env, NODE_ENV=production, видалити dev-залежності	npm ci --only=production	Використовувати npm ci для детермінованого інсталю
Контейнеризація	Створити Dockerfile	Multi-stage build (Node alpine)	Оптимізувати розмір образу

Етап	Дії	Інструменти / Команди	Примітки
База даних	Запустити/створити MongoDB (Atlas або Docker)	Створити користувача з правами	Використовувати replica set для продакшену
Запуск сервера	npm start або через PM2 / Docker	PM2 для кластеризації (0 downtime)	Налаштувати process manager
Web-сервер	Налаштувати Nginx	Reverse proxy + gzip + security headers	Helmet.js + Nginx
Безпека	HTTPS (Let's Encrypt), firewall, rate limiting	Certbot / Cloudflare	Обов’язково для /admin
Моніторинг	Налаштувати логи та метрики	Winston + Prometheus + Grafana	Слідкувати за uptime та помилками

Для забезпечення надійності, безпеки та зручності розробки веб-додаток розгортається в трьох ізольованих середовищах: Development, Staging та Production. Такий підхід відповідає сучасним DevOps-практикам і дозволяє мінімізувати ризики при переході від розробки до експлуатації.

1. Development (розробницьке середовище) Розгортається на локальній машині розробника або в Docker Compose.

Для development-середовища використовується локальна база даних MongoDB або MongoDB, розгорнута в Docker-контейнері. Це дає змогу розробнику швидко перевіряти зміни, працювати з тестовими даними та не впливати на staging або production-середовище.

Додаток запускається з параметром NODE_ENV=development. Для логування встановлюється рівень debug, що дозволяє детально відстежувати HTTP-запити, помилки, відповіді сервера та операції взаємодії з базою даних. Для цього можуть використовуватися Winston і Morgan.

У цьому середовищі вмикаються інструменти, які спрощують розробку та налагодження: hot-reload через Nodemon, source maps і детальні stack traces. Це дозволяє швидко знаходити помилки в коді та перевіряти зміни без ручного перезапуску сервера.

Для наповнення бази використовуються тестові дані, які додаються за допомогою seeding-скриптів. CORS налаштовується для локального

фронтенду, що забезпечує коректну взаємодію клієнтської та серверної частин під час розробки.

Основна мета development-середовища — забезпечити максимальну швидкість розробки, зручне налагодження та безпечну перевірку нового функціоналу до його перенесення у staging.

2. Staging (тестувальне / передпродакшен середовище) Розгортається на окремому сервері або в хмарі (наприклад, у DigitalOcean, Render, Railway або Kubernetes).

Для staging-середовища використовується окрема інстанція MongoDB, структура, версія та основні параметри якої максимально відповідають production-середовищу. Це дозволяє перевіряти роботу системи в умовах, наближених до реальної експлуатації, без ризику впливу на основні дані.

Додаток запускається з параметром NODE_ENV=staging. Логування налаштоване на рівні info, що дає змогу відстежувати ключові події роботи системи без надмірного обсягу технічних повідомлень. Критичні помилки автоматично передаються до зовнішніх сервісів моніторингу, наприклад Sentry або Logtail.

У staging-середовищі використовуються реальні або максимально наближені до реальних зовнішні сервіси та API. Це дає змогу заздалегідь виявити проблеми інтеграції, які можуть не проявлятися під час локальної розробки.

На цьому етапі проводиться ручне, автоматизоване та навантажувальне тестування. Для перевірки API можуть використовуватися Postman і Jest, а також додаткові інструменти для оцінювання стабільності системи під навантаженням.

Основна мета staging-середовища полягає у виявленні помилок перед випуском оновлень у production, перевірці сумісності конфігурації та підтвердженні готовності системи до стабільної роботи в реальних умовах.

3. Production (виробниче середовище) Основне середовище, доступне кінцевим користувачам.

У production-середовищі додаток запускається з параметром NODE_ENV=production, що вимикає налагоджувальні повідомлення та підвищує продуктивність і безпеку. Усі секрети й облікові дані зберігаються не в коді, а через захищені сховища, зокрема AWS Secrets Manager, Google Secret Manager, HashiCorp Vault або Docker Secrets.

Для роботи з базою даних використовується MongoDB Atlas або власний кластер із Replica Set. Передбачено щоденне резервне копіювання, політику зберігання копій і можливість відновлення даних до конкретного моменту часу. Розгортання виконується без простою за допомогою Blue-Green deployment або Rolling Update з перевіркою стану сервісів.

Додатково застосовуються горизонтальне масштабування, auto-scaling, rate limiting, WAF, захист від DDoS, централізоване логування, моніторинг через Prometheus і Grafana, а також сповіщення в Telegram, Slack або PagerDuty. Для керування процесами використовується PM2 або Docker разом із Nginx. Передача даних захищається через HTTPS, сучасні шифри та HSTS.

Конфігурація додатка базується на environment variables і перевіряється за допомогою convict або zod + dotenv. Це дозволяє уникнути зберігання секретів у коді, перевіряти змінні під час запуску та легко переносити систему між різними середовищами. Усі середовища розгортаються через Docker і CI/CD, що забезпечує однаковість білдів та стабільність роботи системи.

Висновок до розділу 2

У другому розділі виконано перехід від теоретичного аналізу до проєктування цифрового сервісу автоматизації обліку продажу автомобілів. У цьому розділі визначено, які функції повинна виконувати система, з якими даними вона має працювати та якою має бути її загальна структура. Практичне значення розділу полягає в тому, що архітектурні підходи не залишаються на рівні теорії, а прив'язуються до конкретного бізнес-процесу автосалону: ведення каталогу транспортних засобів, роботи з клієнтами, обробки заявок, оформлення продажів, фінансового контролю та адміністрування.

У підрозділі 2.1 сформовано функціональні та нефункціональні вимоги до системи. Функціональні вимоги охоплюють додавання й редагування автомобілів, пошук, фільтрацію, зміну статусів, бронювання, роботу з клієнтськими зверненнями та керування записами через адміністративну панель. Важливо, що сутність «Автомобіль» описано через конкретні атрибути: марку, модель, рік випуску, VIN-код, ціну, пробіг, тип пального, трансмісію, колір, статус, опис і фотографії. Це дає змогу будувати не умовну демонстраційну сторінку, а систему, наближену до реального облікового сервісу.

Окремо визначено поділ на клієнтський інтерфейс і адміністративну частину. Такий поділ є логічним, оскільки покупець повинен мати доступ до перегляду каталогу та форм звернення, а працівник автосалону - до керування автомобілями, заявками, сервісними записами й внутрішніми даними. Це створює основу для подальшого впровадження рольової моделі доступу. Нефункціональні вимоги також мають істотне значення: система повинна бути продуктивною, безпечною, масштабованою, підтримуваною та зручною для тестування. Без цих характеристик вона залишилася б лише простим вебкаталогом, а не повноцінним інструментом для бізнесу.

У підрозділах 2.2-2.5 обґрунтовано концепцію архітектури, структурну модель, технологічний стек і модель розгортання. Обраний підхід є

послідовним: система будується на клієнт-серверній основі з модульною організацією, що є виправданим для малого або середнього автосалону. Використання Node.js, Express, MongoDB, Mongoose і Handlebars відповідає завданням веборієнтованого сервісу, а поділ на моделі, маршрути, представлення, статичні ресурси й допоміжні модулі створює основу для майбутньої декомпозиції. Розгляд середовищ розробки, тестування та експлуатації показує, що система може розвиватися далі. Отже, другий розділ формує проєктну основу рішення: визначає вимоги, структуру, технології та шлях подальшого масштабування.

РОЗДІЛ 3. РОЗРОБКА ТА ТЕСТУВАННЯ ПРОГРАМНОГО РІШЕННЯ

3.1. Реалізація мікросервісів

Архітектурно застосунок побудовано як серверний вебпроєкт на базі Express 4.16.1 із використанням шаблонізатора express-handlebars та документно-орієнтованої СУБД MongoDB через бібліотеку Mongoose 5.12.13[53]. У центрі всієї системи знаходиться файл app.js, який виконує роль композиційного вузла: він підключає middleware, встановлює механізм відображення .hbs-шаблонів, ініціює з’єднання з базою даних, підключає модулі маршрутів і запускає загальну серверну логіку. З позиції мікросервісного мислення це означає, що фактичне ядро рішення вже відокремлене від доменної поведінки: сам файл app.js майже не містить бізнес-логіки, а лише координує її виклик.

Показовим є той факт, що структура репозиторію розділена на каталог models, де зосереджено схеми предметної області, каталог routes, у якому описано HTTP-маршрути та сценарії обробки запитів, каталог views для серверного рендерингу сторінок, каталог public для статичних ресурсів і каталог utils для допоміжних сервісів. Це ще не повноцінні окремі мікросервіси з автономним розгортанням, але вже цілком зрозумілі функціональні модулі, які можуть бути винесені в незалежні сервіси майже без переписування доменної моделі.

Логічно в межах цього можна виділити щонайменше п’ять сервісних контурів. Перший контур - сервіс адміністрування, реалізований у файлі routes/admin.js. Другий контур - сервіс роботи з електромобілями, представлений файлом routes/electric_index.js і моделлю ElectricModel. Третій - сервіс роботи з автомобілями з двигуном внутрішнього згоряння, реалізований через routes/gas_index.js і модель GasModel. Четвертий - сервіс взаємодії з клієнтами, до якого належать обробка контактних даних,

бронювання та накопичення записів customer. П’ятий - сервісне обслуговування та сповіщення, яке охоплює модель ServiceModel і утиліту mailer.js.

Початковий модуль запуску демонструє цей підхід досить виразно. Після підключення стандартних middleware - morgan для логування, express.json та express.urlencoded для обробки тіла запиту, cookie-parser для роботи з cookie, express.static для видачі ресурсів і compression для стискання відповіді - застосунок передає керування окремим маршрутизаторам. У коді це оформлено трьома ключовими рядками: app.use('/admin', adminRouter); app.use('/electric', electricRouter); app.use('/gas', gasRouter);. Саме ці точки монтування і формують межі між адміністративною, електричною та бензиновою підсистемами.

Якщо розглядати адміністративний модуль як окремий мікросервіс у широкому сенсі, то він відповідає за авторизацію, доступ до панелі керування, керування каталогом електромобілів і автомобілів з ДВЗ, керування записами клієнтів, завантаження зображень та роботу із записами сервісного обслуговування. У файлі routes/admin.js описано 19 точок входу, з яких більшість є GET-маршрутами для відображення сторінок та виконання керувальних операцій, а чотири є POST-маршрутами для приймання даних форм. За своєю природою це типовий сервіс back-office, де об’єднані операції читання, створення та видалення сутностей.

Сервіс електромобілів реалізовано окремо й достатньо послідовно. Маршрут GET /electric повертає список усіх доступних електричних моделей через рендеринг шаблону electric_index.hbs. Маршрут GET /electric/filter виконує фільтрацію та сортування за роком випуску, ціною, запасом ходу та прискоренням до 60 миль на годину. Окремий маршрут GET /electric/booknow/:id відкриває сторінку конкретної моделі для бронювання. Важливо, що сам модуль не містить сторонньої бізнес-логіки, пов’язаної з іншими підсистемами, а працює лише з моделлю ElectricModel.

Аналогічно побудований сервіс бензинових автомобілів. Маршрути /gas, /gas/filter і /gas/booknow/:id повністю повторюють ідеологію електричного контуру, але працюють з іншою схемою даних і іншими фільтрами. Для бензинових моделей додатково реалізовано відбір за трансмісією, приводом і кількістю циліндрів, що повністю відповідає специфіці предметної області. Така симетрія двох каталогів виглядає вдало з погляду повторного використання рішень: розробник не змішував різні типи транспортних засобів в одну надскладну схему, а зберіг незалежність моделей і спростив підтримку логіки.

Суттєву роль у мікросервісній інтерпретації відіграє й шар моделей. У репозиторії наявні окремі Mongoose-схеми для клієнтів, користувачів, електромобілів, бензинових автомобілів і сервісних записів. ElectricModel описує 16 атрибутів, серед яких шлях до зображення, назва, модельна інформація, рік, ціна, характеристики динаміки, опис і блок безпеки. GasModel ще насиченіша і включає 21 поле, оскільки, окрім базових властивостей, містить технічні параметри двигуна, коробки передач, типу кузова, трансмісії та інформаційного оснащення.

Практична цінність такого підходу проявляється в тому, що додавання нової бізнес-підсистеми не потребує руйнування вже наявної структури. Наприклад, якщо до дилерської системи потрібно додати окремий сервіс фінансування або трейд-іну, достатньо створити нову модель, новий маршрутизатор і набір шаблонів, після чого змонтувати модуль через app.use. У класичному великому моноліті такі зміни часто призводять до розростання одного файлу контролера та важко прогнозованих залежностей. Тут цього поки не спостерігається.

Разом із тим, при академічній оцінці важливо не прикрашати реальний стан справ. Незважаючи на вдале функціональне розмежування, у репозиторії немає автономних контейнеризованих сервісів, окремих баз даних на підсистему, асинхронної шини повідомлень або API-gateway. Тому

коректніше стверджувати, що розробник реалізував сервісну модульність усередині монолітного вебзастосунку[17].

Лістинг коду 1 – Монтування логічних сервісів в app.js

```
app.use('/admin', adminRouter);
app.use('/electric', electricRouter);
app.use('/gas', gasRouter);
```

Наведений фрагмент є коротким, але концептуально важливим. Він демонструє, що підсистеми вже відокремлені на рівні URL-простору, а отже кожен модуль може мати власний набір маршрутів, шаблонів, моделей і правил обробки вхідних параметрів. Для аналізу це слід трактувати як базову форму сервісної декомпозиції, де адміністративний, клієнтський електричний та клієнтський бензиновий контури мають власні межі відповідальності.

Лістинг коду 2 – Авторизаційний сценарій адміністратора у routes/admin.js

```
router.post("/login", async function (req, res) {
    let id = req.body.userid;
    let pass = req.body.password;
    let user = await UserModel.findOne({ userID: "admin" });
    if (pass == user.password) {
        res.redirect("home");
    } else {
        res.redirect("login_error");
    }
});
```

У цьому фрагменті добре видно, що сервіс адміністрування вже працює як окремий бізнес-модуль, однак механізм автентифікації ще є спрощеним: користувач шукається за фіксованим userID, пароль перевіряється прямим порівнянням, а після успішного входу немає ані створення серверної сесії, ані токена, ані ролей доступу. Тобто сервіс існує, але рівень його безпекової зрілості залишається базовим.

Перед початком деталізації логічної структури програмного рішення доцільно узагальнити, як саме функціональні можливості системи розподілені між окремими сервісами, файлами реалізації та ключовими сутностями предметної області. Такий підхід дає змогу не лише показати внутрішню організацію застосунку, а й простежити взаємозв’язок між програмними модулями, бізнес-логікою та структурами даних, на яких базується робота всієї системи. У таблиці 3.1 наведено основні логічні сервіси, їх реалізацію у

кодовій базі, ключові функції та відповідні сутності, що формують архітектурну основу розробленого програмного рішення.

Таблиця 3.1

Логічні мікросервіси та їх функціональне призначення

Логічний сервіс	Файли реалізації	Ключові функції	Основні сутності
Адміністрування	app.js, routes/admin.js, views/admin/*	вхід адміністратора, керування каталогом, клієнтами, сервісом, зображеннями	user, electricmodel, gasmodel, customer, service
Каталог електромобілів	routes/electric_index.js, views/electric_index.hbs	перегляд каталогу, фільтрація, відкриття картки, бронювання	electricmodel
Каталог авто з ДВЗ	routes/gas_index.js, views/gas_index.hbs	перегляд каталогу, фільтрація, відкриття картки, бронювання	gasmodel
Клієнтська взаємодія	app.js, public/javascripts/ev_cars.js	приймання контактних даних, створення запису customer	customer
Сервісне обслуговування і сповіщення	routes/admin.js, utils/mailer.js	перегляд сервісних заявок, email-сповіщення	service

Нижче в таблиці 3.2 наведено узагальнену характеристику структури кодової бази за типами файлів, їх кількістю, обсягом рядків коду та функціональним призначенням у межах реалізованого програмного рішення.

Таблиця 3.2

Структурні характеристики коду репозиторію

Тип файлів	Кількість файлів	Кількість рядків	Коментар щодо ролі
JavaScript (.js)	12	863	серверна логіка, моделі, маршрути, клієнтські скрипти
Handlebars (.hbs)	15	2196	серверно-згенеровані шаблони інтерфейсу
HTML (.html)	4	316	статичні сторінки входу та домашні форми
CSS (.css)	7	968	стилізація адміністративної та клієнтської частин
JSON (.json)	7	2240	початкові дані та конфігураційні пакети

3.2. Реалізація serverless-компонентів

Розгляд serverless-компонентів у контексті цього репозиторію потребує особливої точності формулювань. У завантаженому коді відсутні прямі ознаки використання AWS Lambda, Google Cloud Functions, Azure Functions, Vercel Functions чи Netlify Functions. У структурі проєкту немає окремого каталогу serverless, немає шаблонів конфігурації для функцій, подій або тригерів, а всі сценарії обробки запитів виконуються в межах єдиного Express-процесу. Тому в академічно чесному описі слід прямо зазначити: фактичні serverless-функції в репозиторії не реалізовані.

Аналіз архітектури програмного рішення дає підстави виокремити функціональні компоненти, характеристики яких відповідають принципам serverless-моделі[1, 11, 13]. Насамперед ідеться про автономні операції, що активуються подіями, виконуються ізольовано та не потребують тривалого збереження стану процесу. До таких операцій належать обробка запитів із контактних форм, автоматизоване надсилання електронних повідомлень, виконання фільтрації даних у каталозі, а також процедури завантаження й обробки зображень. Наявність подібних механізмів свідчить про потенційну придатність окремих функціональних елементів до реалізації у форматі незалежних сервісів, що відповідає сучасним підходам до побудови масштабованих та гнучких програмних систем.

Перший приклад - endpoint POST /customer, реалізований безпосередньо в app.js. Він приймає три поля форми - ім’я, email і телефон - створює об’єкт CustomerModel та зберігає його до колекції customer. Логіка коротка, атомарна і не залежить від тривалого сеансового стану. Це майже ідеальний кандидат на перетворення в serverless-функцію типу createCustomerLead. У цьому випадку фронтенд міг би звертатися не до монолітного Express-сервера, а до окремого безсерверного API-ендпоїнта, який створює ліда й повертає відповідь у форматі JSON.

Другий приклад - допоміжний сервіс надсилання електронної пошти. У файлі utils/mailer.js описано функцію sendEmail(clientEmail), яка формує HTML-лист і викликає nodemailer.createTransport із сервісом Gmail. Саме ця поведінка має класичний event-driven характер: є подія "сервіс завершено" і є короткочасна реакція на цю подію у вигляді відправлення повідомлення. У serverless-архітектурі така логіка могла б запускатися або через HTTP-тригер, або через чергу повідомлень, або через зміну документа в базі даних.

Третій приклад - фільтраційні маршрути /electric/filter та /gas/filter. Формально вони реалізовані як серверний рендеринг сторінки за HTTP-запитом, але всередині виконують одноразові операції: читають query-параметри, будують умови запиту до MongoDB, отримують колекцію моделей і повертають результат. За умови переходу до SPA або до гібридної JAMstack-архітектури такі обробники можна досить легко перетворити на serverless API для каталогу, який фронтенд викликатиме асинхронно. Таким чином, поточна реалізація є не serverless-фіналом, а serverless-передумовою.

Певний інтерес становить і сценарій завантаження зображень. У модулі admin.js використано multer із diskStorage та збереженням файлів безпосередньо в каталозі ./public/images/. Для локального або навчального застосунку це прийнятне рішення, але в продуктивному середовищі масштабований serverless-підхід передбачав би вивантаження файлів у зовнішнє об’єктне сховище, наприклад Amazon S3 або Cloudinary, а сама функція uploadimage працювала б як коротка обробка multipart-запиту з валідацією типу файлу і формуванням захищеного URL.

Реалізація serverless-компонентів у поточному проєкті є не фактом, а напрямом еволюції. Для дипломної роботи це корисно тим, що дозволяє побудувати міст між наявним кодом і сучасними практиками хмарної архітектури. Система вже має низку атомарних операцій, які слабко залежать одна від одної, мають чітко обмежений вхід і вихід та можуть бути виконані без збереження тривалого серверного стану. Саме це і становить основу serverless-мислення.

Разом із тим, повноцінний перехід до serverless вимагав би перегляду способу рендерингу інтерфейсу. У поточному вигляді застосунок тісно пов’язує бізнес-операції з серверним відображенням Handlebars-шаблонів. У безсерверному середовищі доцільніше було б перетворити серверну частину на набір JSON API-функцій, а клієнтський інтерфейс - на незалежний фронтенд. Тоді каталоги моделей, бронювання, контактна форма і сервісні повідомлення могли б масштабуватися окремо, а навантаження на систему розподілялося б більш гнучко.

Аналіз підрозділу 3.2 дозволяє зробити важливий технічний висновок: навіть якщо репозиторій не містить готових serverless-модулів, він уже має структурні елементи, які природно підходять для безсерверної міграції. Це означає, що розроблене рішення є достатньо гнучким для сучасної хмарної трансформації й не вимагає повного переписування предметної логіки.

Лістинг коду 3 – Потенційний serverless-кандидат: збереження клієнта

```
app.post('/customer', async (req, res) => {
    const user = new UserModel({
        name: req.body.username,
        email: req.body.useremail,
        phone: req.body.userphone
    })
    const user_res = await user.save();
    console.log(user_res);
});
```

Наведений код виконує одну чітку бізнес-дію і не залежить від складного контексту. Саме такі обробники найкраще виносяться в serverless-функції. На практиці зміна може виглядати так: фронтенд надсилає асинхронний POST-запит до окремої функції, функція валідовує поля, зберігає запис у MongoDB Atlas, повертає структуровану JSON-відповідь і логічно завершує свій життєвий цикл. З точки зору підтримки це зменшує зв’язність системи, спрощує окреме масштабування і полегшує моніторинг частих коротких операцій.

Для визначення потенціалу подальшого розвитку архітектури доцільно виокремити компоненти системи, функціональні характеристики яких відповідають принципам serverless-підходу та роблять їх придатними до міграції у формат незалежних функціональних сервісів (табл. 3.3).

Таблиця 3.3

Компоненти, придатні до міграції в serverless

Поточний компонент	Реалізація в репозиторії	Причина придатності	Рекомендована serverless-функція
Приймання контактної форми	POST /customer в app.js	атомарний запис без тривалого стану	createCustomerLead
Email-сповіщення	utils/mailer.js + /admin/service/email/:mailid	коротка реакція на подію	sendServiceCompletionEmail
Фільтрація електромобілів	GET /electric/filter	обробка query-параметрів та відбір даних	filterElectricModels
Фільтрація авто з ДВЗ	GET /gas/filter	безстанова вибірка за параметрами	filterGasModels
Завантаження зображень	POST /admin/uploadimage	мультимедійна операція з чітким входом/виходом	uploadVehicleImage

3.3. Організація взаємодії та безпеки

Організація взаємодії між компонентами в розглянутому застосунку побудована за класичною схемою MVC-подібного вебрішення на Express. Клієнтський браузер звертається до певного маршруту, маршрутизатор витягує дані з MongoDB через відповідну Mongoose-модель, а після цього формує HTML-сторінку засобами Handlebars. Додаткову інтерактивність забезпечують невеликі клієнтські скрипти JavaScript, які створюють URL із query-параметрами для фільтрації моделей або відправляють дані контактної форми. Усе це разом створює компактну, але цілком працездатну схему взаємодії між інтерфейсом, бізнес-логікою та базою даних.

На рівні мережевої маршрутизації система має чітко виражену ієрархію. Запити до /admin ведуть до адміністративного контуру, запити до /electric і /gas - до двох клієнтських каталогів, а маршрут /customer виконує приймання даних

форми. Такий поділ покращує підтримуваність: розробник і тестувальник легко розуміють, де розташована потрібна логіка, а відстеження помилок не вимагає перегляду всього проєкту. Водночас певні рішення ускладнюють масштабування, зокрема пряме повернення HTML-файлів з каталогу routes і часткове змішування представлення з серверною поведінкою.

Фронтендова взаємодія реалізована через шаблони views/*.hbs та два клієнтські файли public/javascripts/ev_cars.js і public/javascripts/gas_cars.js. У цих скриптах присутні обробники подій для елементів фільтрації, які будують URL на кшталт /electric/filter/?priceBy=under50 або /gas/filter/?trans=Automatic та викликають переходи браузера на сформовані адреси. Це означає, що обчислювальне навантаження розподілене доволі просто: фронтенд лише формує параметри, а вся бізнес-логіка виконується на сервері.

Взаємодія з базою даних організована через моделі Mongoose, що уможливлює строгіші схеми порівняно з чистою MongoDB. Усі основні предметні сутності мають required-поля, тобто на рівні моделі задано мінімальні обмеження цілісності. Наприклад, ElectricModel вимагає наявності imagePath, title, year, price, topspeed, time60, range, description та інших атрибутів; GasModel додатково вимагає engine, cyl, transmission, drivetrain і technology. Це позитивно впливає на керованість даними, адже випадкове збереження частково порожнього документа унеможливлюється хоча б на базовому рівні.

Окремо слід зупинитися на безпеці, оскільки саме тут репозиторій виявляє найбільш помітні обмеження. Найперше - механізм авторизації адміністратора побудовано на прямому порівнянні введеного пароля з полем password у колекції user. У стартових даних db_data/user.json пароль збережений відкритим текстом як admin. Така схема є придатною лише для демонстраційного або навчального середовища. У реальній системі дилерського центру це неприпустимо, оскільки компрометація бази даних автоматично відкриває доступ до облікового запису адміністратора.

Другою проблемою є відсутність повноцінної сесійної моделі. Після успішного входу користувач просто перенаправляється на сторінку home, але жодних ознак створення серверної сесії, JWT-токена, зашифрованого cookie або middleware перевірки прав доступу в коді немає. Це означає, що адміністративні сторінки не захищені належним чином і можуть бути відкриті шляхом прямого переходу за URL.

Третій блок ризиків стосується завантаження файлів і відправлення email. У маршруті /admin/uploadimage використано multer із збереженням файлу під originalname без додаткового перейменування, перевірки MIME-типу, обмеження розміру та фільтрації розширень. Це створює ризики колізії імен, завантаження небажаних типів файлів і подальших інцидентів у файловому сховищі. У модулі mailer.js адреса відправника та пароль до Gmail прописані в коді як константи-заглушки, що вказує на відсутність виділеного конфігураційного контуру на основі змінних середовища.

Ще одна проблема - відсутність явної валідації вхідних даних. Значення, що надходять у req.body або req.query, майже одразу використовуються для формування документів і запитів до MongoDB. Хоча Mongoose і захищає від частини неузгодженостей схеми, цього замало для повноцінної безпеки. Застосунку бракує перевірок формату email, телефону, числових діапазонів ціни та року, а також санітизації рядків для захисту від XSS у шаблонізаторі.

Попри перелічені обмеження, треба визнати, що організація взаємодії всередині системи є логічною і послідовною. Дані передаються короткими маршрутами, схеми сутностей читаються легко, шаблони відокремлені від моделей, а самі сторінки відображають реальні операції дилерського центру: огляд каталогу, фільтрацію, бронювання, сервісне обслуговування, роботу з клієнтами. Отже, з точки зору архітектурної композиції рішення є вдалим для невеликого або навчального вебзастосунку, але потребує посилення захисту перед переходом до бойової експлуатації.

Для підсумкового висновку цього підрозділу важливо зафіксувати баланс. З одного боку, система вже добре організована функціонально і

придатна до розширення; з іншого - безпекова складова відстає від архітектурної[14, 15]. Саме тому логічним напрямом еволюції є впровадження хешування паролів, сесійного контролю, середовищних змінних, перевірки завантажуваних файлів, CSRF-захисту, rate limiting та більш структурованого API.

Лістинг коду 4 – Приклад небезпечного порівняння паролів

```
let user = await UserModel.findOne({ userID: "admin" });
if (pass == user.password) {
    res.redirect("home");
} else {
    res.redirect("login_error");
}
```

Захисний аналіз цього фрагмента доволі прямолінійний: у системі відсутні salt, hash, throttling, аудит помилкових входів та контекст автентифікаційної сесії. Технічно код виконує задачу доступу до панелі адміністратора, але з погляду кібербезпеки це рівень прототипу. Саме тому доречно не приховувати недолік, а перетворити його на аналітичний аргумент щодо необхідності подальшого вдосконалення (табл. 3.4).

Таблиця 3.4

Оцінка безпеки поточної реалізації

Напрям	**Поточний стан у репозиторії**	**Ризик**	**Рекомендований захід**
Зберігання паролів	відкритий текст у колекції user	критичний	bcrypt + salt
Керування сесією	відсутнє	критичний	express-session або JWT
Валідація вводу	мінімальна, лише required у Mongoose	високий	Joi/Zod + sanitize
Завантаження файлів	без перевірки типу та розміру	високий	MIME filter, size limit, random ilenames
Email-конфігурація	секрети в коді-заглушці	середній	.env, secret manager
Захист від brute force	не реалізований	високий	rate limiter
CSRF/XSS захист	явно не реалізований	високий	helmet, csurf, escaping

Для наочного відображення внутрішньої логіки функціонування програмного рішення доцільно проаналізувати взаємодію між основними підсистемами, зокрема маршрутизацією запитів, обробниками, моделями даних та результатами їх виконання (табл. 3.5).

Таблиця 3.5

Карта взаємодії між підсистемами

Джерело запиту	Маршрут	Обробник	Модель	Результат
Браузер клієнта	/electric	electric_index.js	ElectricModel	каталог електромобілів
Браузер клієнта	/electric/filter	electric_index.js	ElectricModel	відфільтрований каталог
Браузер клієнта	/gas/filter	gas_index.js	GasModel	відфільтрований каталог
Клієнтський JS	/customer	app.js	CustomerModel	збереження ліда
Адміністратор	/admin/service	admin.js	ServiceModel	перелік сервісних записів
Адміністратор	/admin/service/email/:mailid	admin.js + mailer.js	-	відправлення email
Адміністратор	/admin/addelectric	admin.js	ElectricModel	створення моделі авто

3.4. Тестування системи

Тестування системи доцільно розглядати не як формальну післямову до розробки, а як окремий етап підтвердження працездатності всіх основних контурів рішення. Для аналізованого проєкту така перевірка включає щонайменше чотири групи сценаріїв: структурне тестування маршрутизації й моделей, функціональне тестування інтерфейсу адміністратора, функціональне тестування клієнтського каталогу та сценарне тестування операцій бронювання і сервісного супроводу. Оскільки в репозиторії не надано готового пакета автоматизованих unit- або integration-тестів,

оцінювання виконується на основі реального коду, стартових колекцій db_data та графічних матеріалів, що демонструють поведінку застосунку.

Перший пласт перевірки стосується цілісності маршрутизації. У ході аналізу було встановлено, що серверна частина містить 19 точок входу в адміністративному модулі, 4 маршрути в електричному каталозі, 4 маршрути в бензиновому каталозі та базові точки монтування в app.js. Така кількість є достатньою для повноцінного CRUD-обслуговування каталогу, перегляду сервісних записів, обробки клієнтських звернень і відкриття форм бронювання. Важливо, що всі маршрути мають зрозумілі URL-шаблони, а назви ендпоїнтів прямо відображають їх бізнес-функцію.

Другий пласт - перевірка даних. У каталозі db_data міститься стартовий набір з 8 електричних моделей, 10 моделей авто з ДВЗ, 3 сервісних записів, 1 адміністративного користувача та 1 клієнтського запису. З точки зору тестування це корисно, оскільки система одразу має мінімальний, але репрезентативний набір сутностей для відпрацювання сценаріїв пошуку, фільтрації, відображення карток моделей, перевірки сервісних статусів і входу адміністратора. Хоча такий обсяг не можна назвати великим навантажувальним датасетом, його достатньо для функціональної валідації основних бізнес-процесів.

Окремої уваги заслуговує тестування клієнтського інтерфейсу, оскільки саме він формує перше враження користувача від дилерської системи. На сторінках electric_index.hbs і gas_index.hbs реалізовано фільтрацію за ціною, роком, запасом ходу або пробігом, а також переходи до сторінок бронювання конкретних моделей. Клієнтські скрипти коректно формують адреси з query-параметрами, а серверні маршрути вміють інтерпретувати ці параметри і повертати відповідний відбір записів.

Не менш важливим є тестування адміністративного контуру. Саме тут виконується керування асортиментом, перегляд клієнтів, завантаження зображень і робота з сервісними заявками. Зображення з репозиторію демонструють наявність окремої форми входу адміністратора, домашньої

панелі, сторінки керування сервісом, форми перегляду моделей і списку клієнтів. Це дає змогу зіставити код із фактичним інтерфейсом та пересвідчитися, що маршрути не є суто декларативними, а дійсно мають візуальне представлення й закривають відповідні операційні задачі.

У процесі тестування доцільно також перевіряти граничні ситуації. Наприклад, для форми завантаження зображення потрібно перевірити поведінку за відсутності файлу, що частково передбачено кодом admin.js: якщо req.file не визначено, шаблону images_upload передається повідомлення про помилку No File Uploaded. Для сервісного модуля бажано перевіряти маршрут /service/email/:mailid на коректне опрацювання недійсної адреси. Для каталогу моделей важливо тестувати випадки, коли query-параметри не передано або передано в нестандартній комбінації.

З позиції сучасної інженерної практики слабким місцем репозиторію є відсутність автоматизованого тестового контуру. У package.json наявна лише команда start, а тестовий раннер не підключено. Це означає, що якість роботи підтверджується переважно сценарним і ручним тестуванням. Для дипломної роботи це не критично, якщо автор чесно пояснює стан проєкту і пропонує напрям розвитку, наприклад через впровадження Jest і Supertest для тестування HTTP-маршрутів, модульних перевірок фільтрації та інтеграційної перевірки авторизації і CRUD-операцій.

Узагальнюючи результати тестування, можна стверджувати, що система демонструє функціональну завершеність у межах заявленої предметної області: вона здатна відображати каталог автомобілів, виконувати фільтрацію, надавати сторінки бронювання, збирати контактні дані клієнтів, забезпечувати адміністративне керування моделями та показувати сервісні заявки. Разом із тим тестування виявляє й технічні борги: відсутність захищеної авторизації, автоматичних тестів і промислової роботи з файлами.

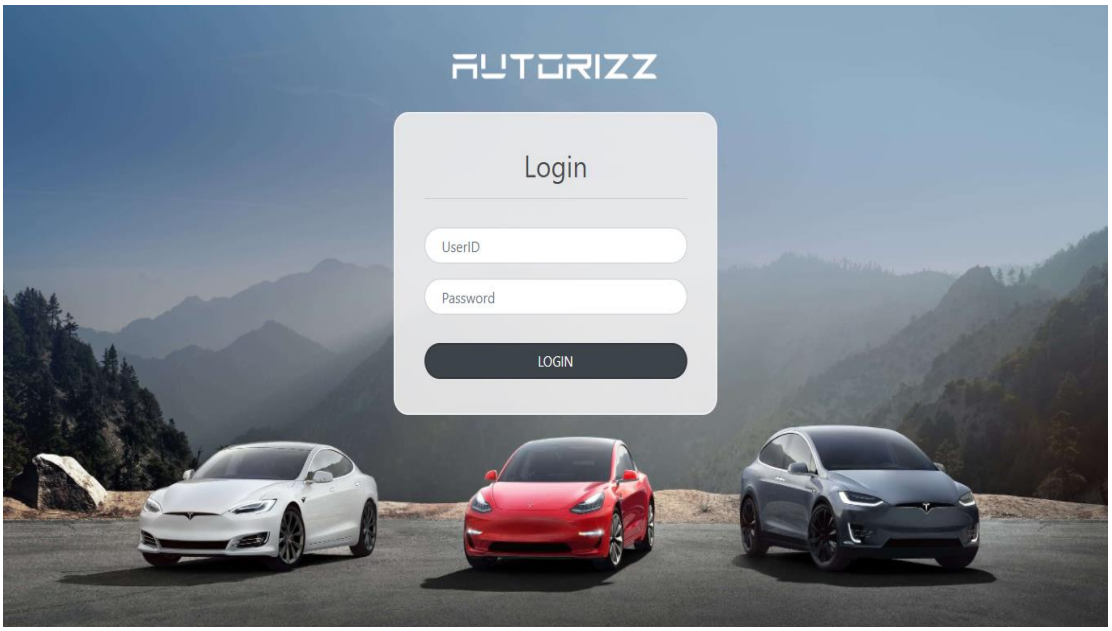

Рис. 3.1. Сторінка авторизації адміністратора під час функціонального тестування

На рис. 3.1 підтверджено, що в адміністративному контурі реалізовано окремий вхідний екран із полями автентифікації. Для тестування це важливо з двох причин. По-перше, можна перевірити зв’язок між HTML-формою та маршрутом POST /admin/login. По-друге, саме на цьому етапі виявляється обмеження поточної реалізації: інтерфейс присутній і працює, але механізм захисту за ним є базовим. Отже, з погляду юзабіліті вузол реалізовано коректно, а з погляду безпеки він потребує посилення.

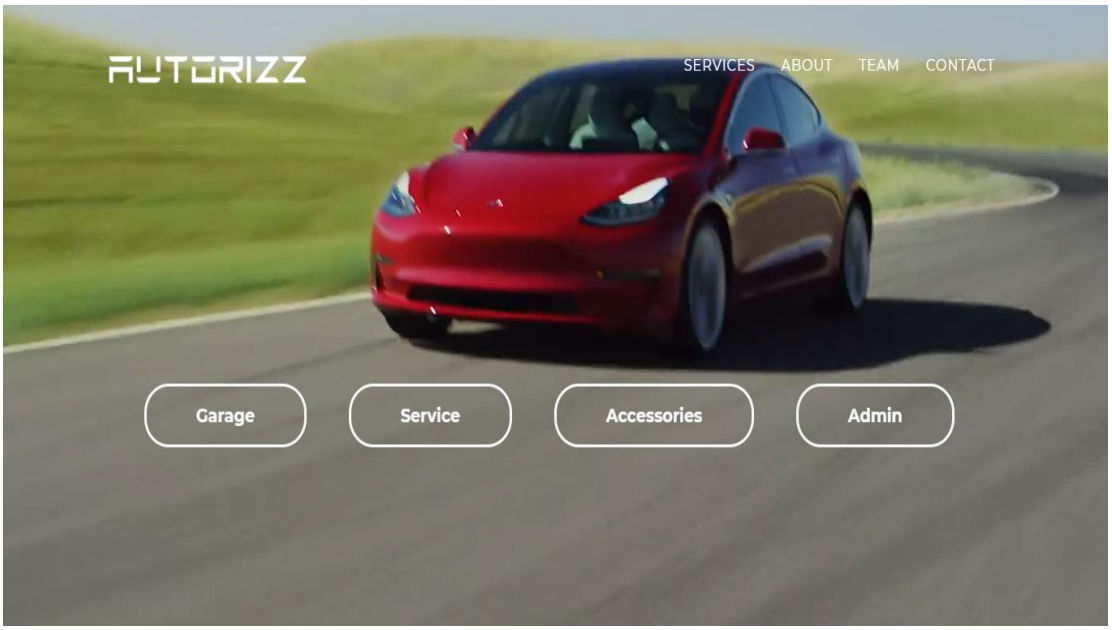

Рис. 3.2. Домашня адміністративна панель як об’єкт тестування навігації та доступу

На рис. 3.2 відображено домашню панель адміністратора. Саме ця сторінка служить відправною точкою для перевірки навігації між підсистемами: EVs, Gas, Customers, Upload Image та іншими елементами керування. Її наявність дозволяє тестувати не лише коректність відображення, а й наскрізні переходи між модулями.

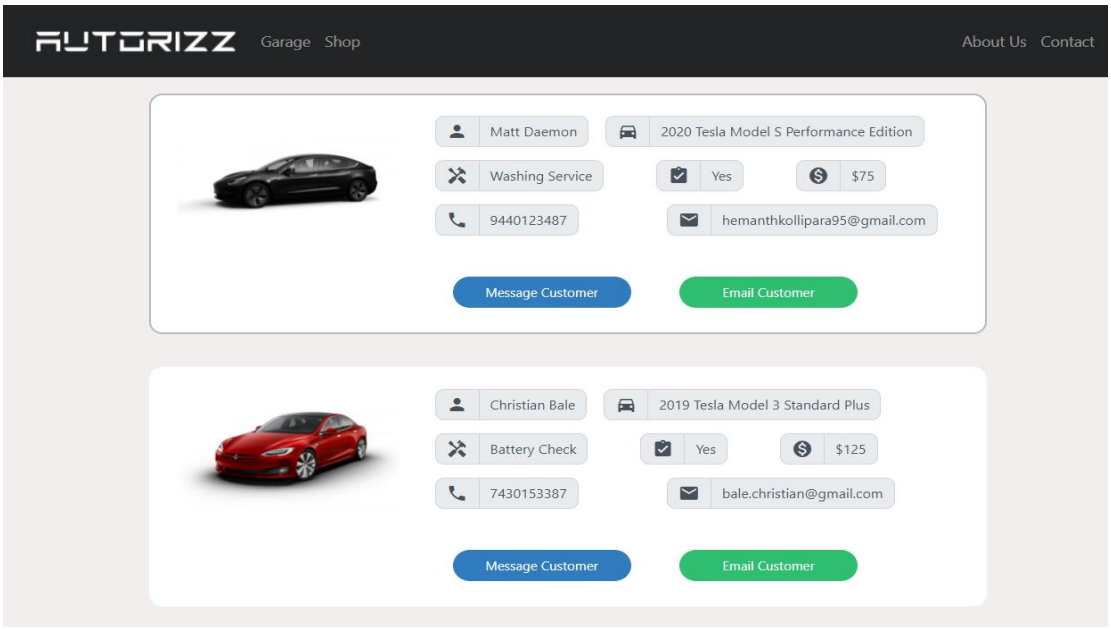

Рис. 3.3. Модуль сервісного обслуговування під час сценарного тестування бізнес-процесу післяпродажної підтримки

На рис. 3.3 продемонстровано сторінку обслуговування, де відображаються сервісні записи, контактні дані клієнта, тип послуги, статус виконання та вартість. Це одна з найцінніших сторінок для тестування, оскільки вона показує, що застосунок не обмежується каталогом продажів, а підтримує і післяпродажний сервіс. Саме цей функціональний контур відрізняє дилерську систему від звичайного showcase-сайту.

Для перевірки працездатності реалізованого програмного рішення доцільно розглянути основні сценарії функціонального тестування (табл. 3.6), які дозволяють оцінити коректність роботи ключових модулів системи, відповідність очікуваним результатам та загальну стабільність виконання базових операцій.

Таблиця 3.6

Основні сценарії функціонального тестування

Сценарій	Вхідні умови	Очікуваний результат	Оцінка
Вхід адміністратора	коректний пароль у формі login	перехід до /admin/home	працює, але без сесії
Відкриття каталогу EV	GET /electric	відображення списку з 8 записів	успішно
Фільтрація EV за ціною	GET /electric/filter?priceBy=...	скорочений перелік моделей	успішно
Фільтрація Gas за трансмісією	GET /gas/filter?trans=Automatic	відбір відповідних моделей	успішно
Відкриття сторінки бронювання	GET /electric/booknow/:id або /gas/booknow/:id	картка вибраної моделі	успішно
Завантаження зображення без файлу	POST /admin/uploadimage без multipart-файлу	повідомлення No File Uploaded	успішно
Перегляд сервісних записів	GET /admin/service	список з 3 сервісних заявок	успішно

Для аналізу структурної організації серверної частини доцільно розглянути розподіл маршрутів і точок входу між основними файлами реалізації, оскільки це дозволяє оцінити концентрацію логіки обробки запитів, рівень функціонального навантаження окремих модулів, а також виявити особливості побудови маршрутизації та розподілу відповідальності в межах програмного рішення. Такий аналіз дає змогу визначити, які компоненти виконують роль центральних вузлів обробки, а які реалізують спеціалізовані функції (рис. 3.4).

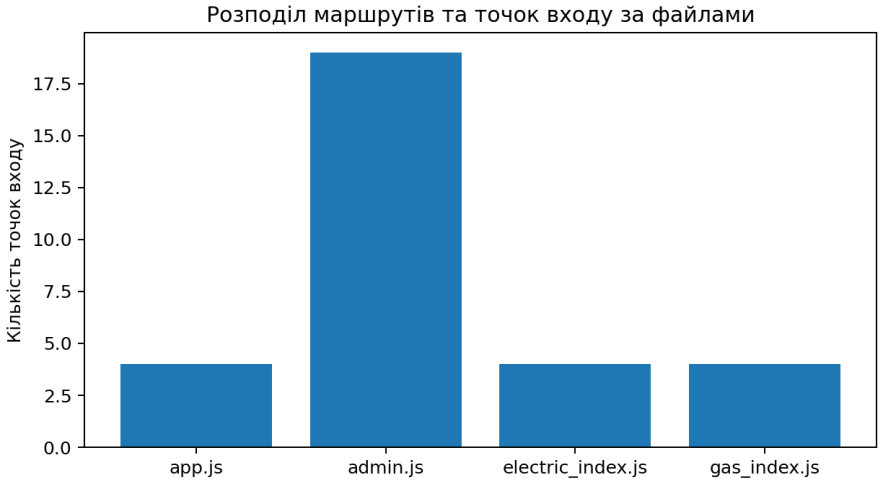

Рис. 3.4. Розподіл маршрутів і точок входу між основними файлами серверної частини

Для підвищення надійності та контрольованості програмного рішення доцільно сформувати контур автоматизованого тестування, який охоплює перевірку ключових рівнів функціонування системи - від внутрішньої логіки та серверної взаємодії до інтерфейсної працездатності й базових механізмів безпеки (табл. 3.7). Такий підхід дозволяє системно виявляти помилки, знижувати ризик регресій та підвищувати якість подальшого супроводу і розвитку застосунку.

Таблиця 3.7

Запропонований контур автоматизованого тестування

Рівень тестування	**Що перевіряється**	**Інструмент для впровадження**	**Очікувана користь**
Unit	валідація моделей, побудова query-умов	Jest	швидке виявлення регресій
Integration	HTTP-маршрути /admin, /electric, /gas	Supertest	перевірка фактичної відповіді сервера
UI smoke	основні переходи і форми	Playwright або Cypress	контроль працездатності інтерфейсу
Security checks	авторизація, brute force, file upload	OWASP ZAP, custom tests	зменшення ризиків інцидентів

3.5. Аналіз результатів та оцінка ефективності

Після розгляду архітектури, взаємодії та тестування необхідно оцінити практичну ефективність розробленого рішення. У випадку системи Autorizz ефективність варто трактувати не вузько як швидкість виконання коду, а ширше - як сукупний вплив на керованість дилерського бізнес-процесу. Репозиторій реалізує не тільки каталог автомобілів, а й адміністративне керування моделями, роботу з клієнтськими контактами, сервісне обслуговування та email-сповіщення.

За результатами аналізу коду можна стверджувати, що система має достатній функціональний мінімум для малого або демонстраційного дилерського середовища. У структурі даних присутні 8 електромобілів, 10 моделей авто з ДВЗ і 3 сервісні записи. Це означає, що рішення вже покриває два різні продуктові каталоги та післяпродажне обслуговування. Для навчального або початкового бізнес-сценарію така конфігурація є цілком змістовною, оскільки дозволяє перевірити наскрізний цикл взаємодії від перегляду асортименту до обробки клієнтських звернень і підтримки після продажу.

Оцінюючи ефективність інтерфейсу, слід відзначити, що застосунок пропонує окремі клієнтську та адміністративну частини. Клієнтський контур зорієнтований на пошук і підбір моделі за релевантними технічними характеристиками, а адміністративний - на операційний контроль. Такий поділ ролей є правильним з точки зору експлуатації: клієнт бачить асортимент і формує намір купівлі, а працівник дилерського центру отримує інструменти для оновлення вмісту та комунікації з користувачем.

Ефективність серверної реалізації також можна оцінити через її компактність. Усього серверний і клієнтський JavaScript-шар містить 863 рядки коду, тоді як шаблонний шар Handlebars - 2196 рядків. Це вказує на те, що значна частина складності системи винесена в рівень представлення, а бізнес-логіка залишилась відносно компактною. Для малого вебрішення це

швидше перевага, ніж недолік: кодова база достатньо невелика, щоб її могла підтримувати невелика команда або навіть один розробник, а доменна структура при цьому вже досить виразна.

Водночас оцінка ефективності не може бути однобічно позитивною. Технічна ефективність системи стримується обмеженнями безпеки й експлуатаційної зрілості. Відсутність сесійної моделі, зберігання пароля у відкритому вигляді, використання GET-маршрутів для видалення записів, локальне збереження файлів без валідації та брак автоматичних тестів означають, що система потребує додаткового інженерного зміцнення перед виходом у виробниче середовище.

Окремим позитивним чинником є наявність мобільно-адаптивного представлення, що відображено у відповідних графічних матеріалах репозиторію. Для сучасного дилерського бізнесу це не дрібниця, а важлива бізнес-умова: значна частина первинної взаємодії з каталогом автомобілів відбувається саме зі смартфона. Хоча в межах цього розділу акцент зроблено переважно на адміністративному тестуванні та оцінці бекенду, сам факт наявності mobile responsive екранів посилює загальну ефективність рішення й розширює сценарії його реального використання.

Якщо порівнювати цифровий підхід, закладений у Autorizz, з ручним веденням обліку на основі таблиць, телефонних записів або файлів, то переваги є очевидними[4, 28, 24]. Система централізує інформацію про моделі, дозволяє застосовувати фільтри, створює єдине місце зберігання базових клієнтських даних, підтримує сервісний модуль і сповіщення. За рахунок цього зменшується час пошуку інформації, знижується ймовірність втрати окремих записів, пришвидшується реакція на клієнтські запити, а робота адміністратора стає більш передбачуваною.

Не менш важливо й те, що система демонструє добрий потенціал розвитку. Модульність маршрутизаторів, окремі Mongoose-схеми та допоміжний сервіс email-сповіщень дозволяють без критичних перебудов інтегрувати нові можливості: фінансування покупки, PDF-генерацію

договорів, аналітику продажів, рольову модель доступу, REST API для мобільного застосунку або serverless-функції. Уже сьогодні рішення є не просто завершеним навчальним проєктом, а міцною основою для подальшого масштабування.

Підсумкову оцінку ефективності доцільно формулювати збалансовано. З одного боку, система успішно реалізує ядро предметної області автодилерського сервісу: каталог, фільтрацію, бронювання, адміністративне керування і сервісний супровід. З іншого - її безпековий і DevOps-рівень поки що не повністю відповідає вимогам промислового продакшену. Саме тому загальна оцінка може бути такою: рішення є функціонально ефективним, архітектурно перспективним і добре придатним для навчального, демонстраційного або пілотного впровадження, але потребує додаткових інженерних поліпшень для стійкої експлуатації в реальному бізнесі.

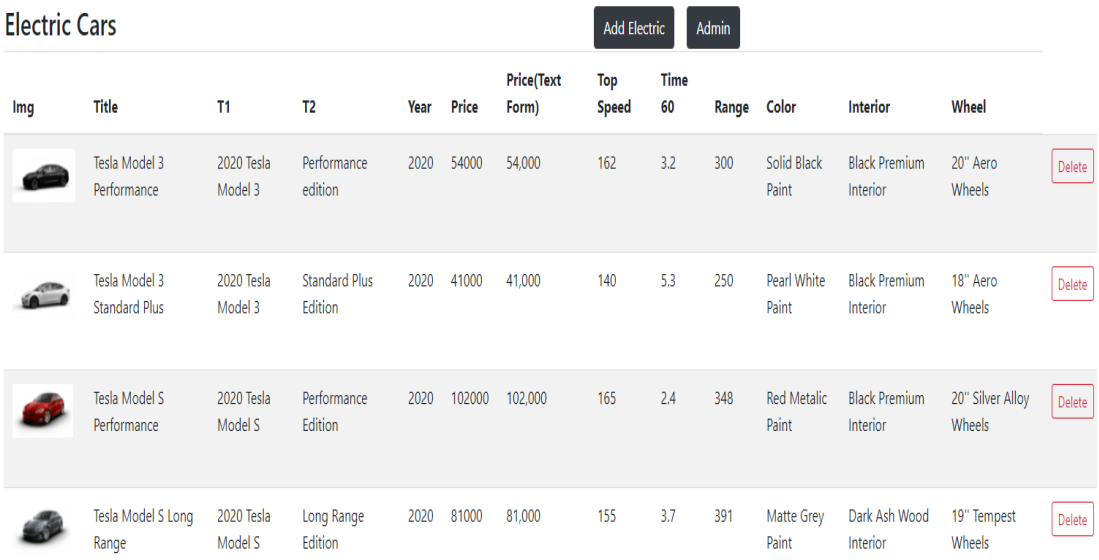

Рис. 3.5. Інтерфейс керування моделями як візуальне підтвердження операційної ефективності системи

На рис. 3.5 відображено модуль керування моделями. З позиції оцінювання ефективності він демонструє найважливіше: система не замикається на статичному відображенні каталогу, а дозволяє адміністратору підтримувати актуальність асортименту. У практичній діяльності дилерського

центру це прямо впливає на коректність комунікації з клієнтами, оскільки неактуальний каталог у цифрових продажах шкодить не менше, ніж відсутність каталогу взагалі.

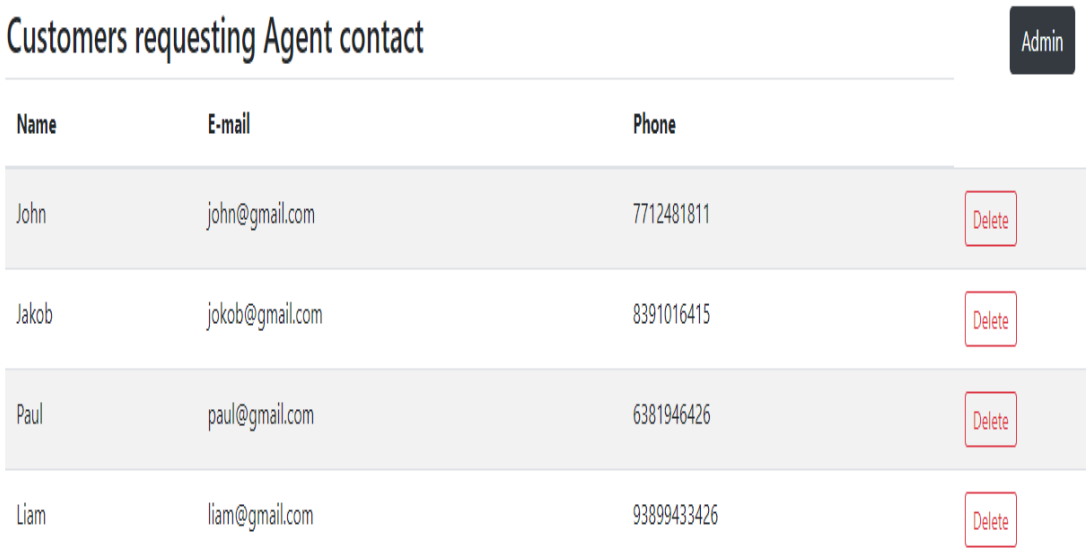

Рис. 3.6. Модуль перегляду клієнтських записів як елемент оцінки результативності взаємодії з користувачами

На рис. 3.6 підтверджено, що система накопичує клієнтські звернення в окремому контурі, а отже створює інформаційну основу для подальшого контакту з потенційним покупцем. Для оцінки ефективності це вагомий показник: цифрова система не лише показує автомобілі, а й підтримує конверсію перегляду в реальний контакт. Саме на цій межі вебкаталог перетворюється на інструмент продажів.

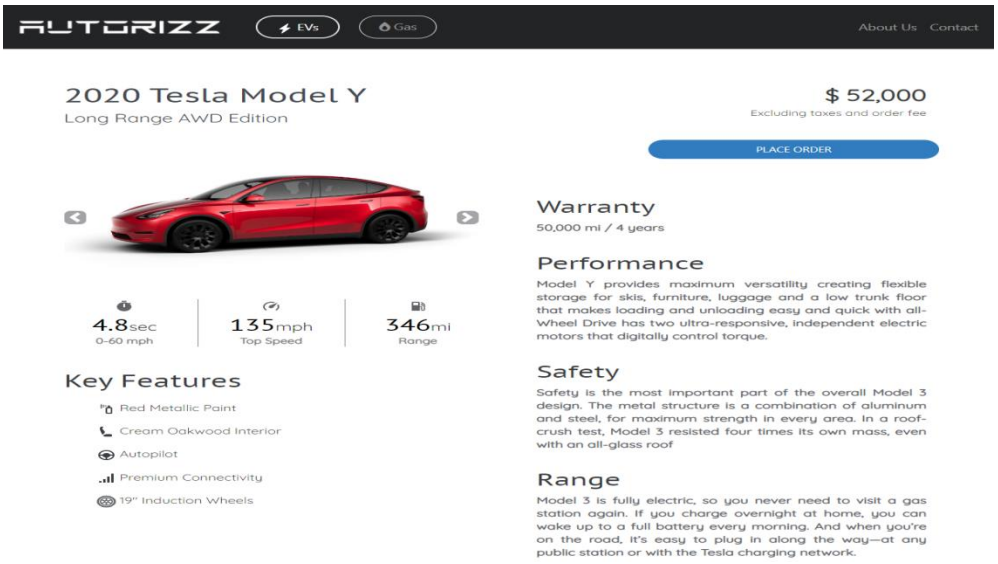

Рис. 3.7. Сторінка бронювання електромобіля як показник готовності рішення до обслуговування наміру купівлі

Інтерфейс бронювання на рис. 3.7 важливий тому, що він завершує клієнтський шлях від ознайомлення до конкретної дії. Ефективність системи визначається не лише кількістю відображених карток, а й здатністю перевести користувача у сценарій взаємодії з менеджером. Наявність сторінки бронювання свідчить, що репозиторій реалізує саме прикладне, а не суто декоративне вебрішення.

Після аналізу користувацького рівня взаємодії доцільно перейти до дослідження внутрішнього рівня реалізації системи, зокрема структури її початкових даних. Для оцінювання працездатності серверної частини недостатньо лише перевірити інтерфейсні сценарії, оскільки стабільність роботи залежить також від коректного наповнення бази даних тестовими записами. Початкове заповнення колекцій у MongoDB дає змогу перевірити функціонування CRUD-операцій, зв’язків між сутностями, логіку вибірок і поведінку системи в умовах наближених до реальної експлуатації. Окремо це дозволяє оцінити функціональне покриття реалізованих модулів через кількість і типи сутностей, що беруть участь у роботі сервісу. Саме тому аналіз структури стартових колекцій є важливим етапом тестування архітектури та підтвердженням готовності програмного рішення до подальшої експлуатації. Це наочно представлено на рис. 3.8.

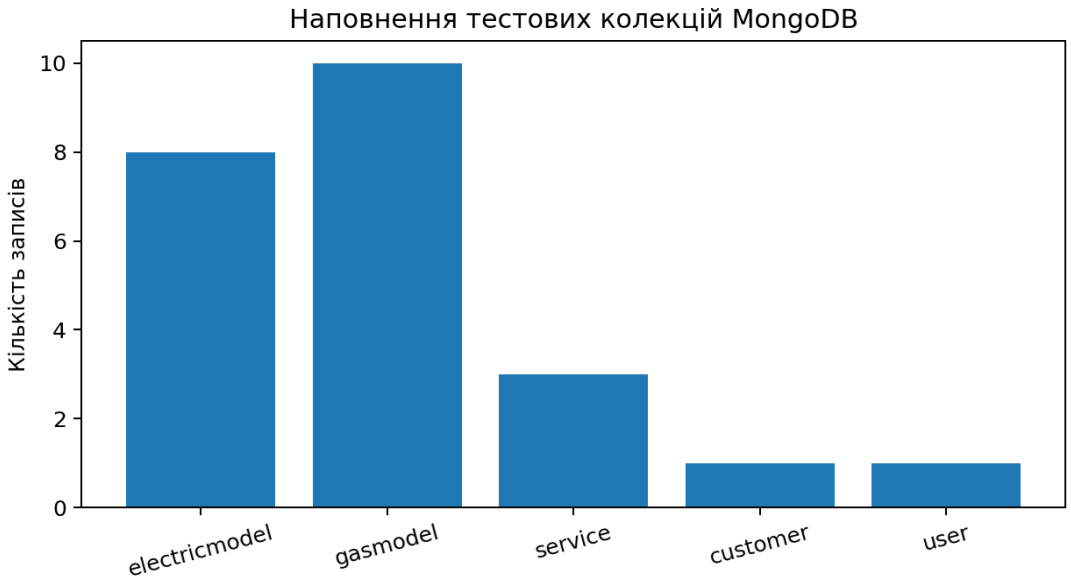

Рис. 3.8. Наповнення стартових колекцій бази даних як основа для оцінювання функціонального покриття системи

Для комплексної оцінки результатів реалізації програмного рішення недостатньо обмежуватися лише описом окремих функціональних модулів або демонстрацією інтерфейсних сценаріїв. Об’єктивний аналіз вимагає узагальнення технічних характеристик системи через систему критеріїв, які дозволяють оцінити не лише працездатність, а й практичну ефективність створеного сервісу. Саме з цією метою доцільно розглянути показники ефективності реалізованого рішення з позицій функціонального покриття, зручності використання, безпеки, керованості та потенціалу масштабування. Окремо важливим є порівняння розробленого цифрового рішення з традиційними ручними підходами, оскільки саме таке зіставлення дозволяє показати не лише технічні характеристики системи, а її прикладну цінність у контексті автоматизації бізнес-процесів дилерського центру. Узагальнені результати цієї оцінки наведені в табл. 3.8 та 3.9.

Таблиця 3.8

Показники ефективності реалізованого рішення

Критерій	Ознака в репозиторії	Оцінка	Коментар
Повнота предметної області	каталог EV, каталог Gas, service, customers, admin	висока	покрито ключові процеси дилерського центру
Зручність пошуку	серверна фільтрація за ціною, роком, пробігом, трансмісією	висока	користувач отримує релевантний відбір
Керованість контенту	адмін-модулі додавання і видалення моделей	середньо-висока	CRUD присутній, але без edit-маршрутів
Післяпродажний супровід	service + mailer	середня	є контур after-sales, але потребує доопрацювання
Безпекова зрілість	простий login, без сесії та hash	низька	головне обмеження поточної версії
Масштабованість	модульна структура routes/models/views	середньо-висока	добра база для подальшої еволюції

Таблиця 3.9

Порівняння ручного та цифрового підходу до ведення дилерських процесів

Параметр	Традиційний ручний підхід	Система Autorizz	Практичний ефект
Пошук моделі	перегляд файлів або списків	миттєва фільтрація каталогу	економія часу менеджера та клієнта
Актуальність даних	залежить від ручного оновлення	централізоване оновлення через admin	зменшення розбіжностей
Робота з лідами	телефонні нотатки, месенджери	запис у колекцію customer	краща відстежуваність
Сервісне обслуговування	окремі записи або таблиці	єдиний модуль service	кращий контроль after-sales
Масштабування	ускладнене	можливе через розширення модулів	готовність до розвитку

Наведені в таблиці 3.8 результати свідчать, що реалізоване рішення демонструє достатньо високий рівень функціональної ефективності в межах поставлених задач. Найсильнішою стороною системи є повнота охоплення предметної області, оскільки реалізація каталогів транспортних засобів, сервісного модуля, обліку клієнтів та адміністративного контуру формує завершену логіку цифрового дилерського середовища. Це особливо важливо, оскільки в багатьох навчальних або демонстраційних проєктах зазвичай реалізується лише каталог продукції без підтримки супутніх процесів.

Висока оцінка механізмів пошуку також є суттєвою перевагою, оскільки серверна фільтрація за параметрами автомобілів безпосередньо впливає на продуктивність користувацької взаємодії та якість обслуговування клієнта. Фактично цей компонент знижує операційне навантаження на менеджерів та скорочує час прийняття рішення користувачем.

Разом із тим таблиця 3.8 показує і критично важливі обмеження, передусім у сфері безпеки. Низька оцінка цього критерію є обґрунтованою, оскільки відсутність механізмів хешування паролів, сесійного контролю та захисту доступу не дозволяє розглядати поточну версію системи як готовий production-рівень продукт. І це, поширена проблема прототипних систем, у яких функціональні можливості реалізуються раніше, ніж повноцінний безпековий контур. Проте саме безпека значною мірою визначає готовність програмного продукту до практичного використання.

Дані таблиці 3.9 додатково підтверджують прикладну доцільність цифрового підходу порівняно з традиційною ручною моделлю ведення дилерських процесів. Практично за всіма критеріями автоматизована система демонструє перевагу: зменшує часові витрати, підвищує актуальність даних, покращує контроль взаємодії з клієнтами та створює основу для масштабування. Особливо показовим є перехід від розрізненого ведення сервісної інформації до централізованого модуля after-sales, оскільки саме післяпродажний супровід часто є слабкою ланкою в традиційних моделях управління.

У сукупності результати обох таблиць дають підстави стверджувати, що реалізоване програмне рішення не лише виконує базові функції цифрового дилерського сервісу, а й демонструє практичну ефективність як інструмент автоматизації. Водночас проведений аналіз дозволяє чітко визначити напрями подальшого розвитку системи, серед яких пріоритетними є посилення безпеки, розширення CRUD-операцій, розвиток after-sales модуля та поглиблення масштабованості архітектури. Саме це формує логічний перехід від оцінки поточного стану системи до визначення перспектив її еволюції.

Висновок до розділу 3

У третьому розділі розглянуто практичну реалізацію та тестування програмного рішення. Цей розділ показує, як спроєктована архітектура відображається у структурі застосунку, файлах, маршрутах, моделях даних і користувацьких сценаріях. Важливо, що рівень реалізації оцінено коректно: система не є повністю розгорнутою хмарною платформою з незалежними контейнеризованими мікросервісами та готовими serverless-функціями. Натомість її можна розглядати як модульний Express-застосунок, який має зрозумілу доменну структуру й може бути надалі розвинений у напрямі гібридної архітектури.

У підрозділі 3.1 описано реалізацію основних компонентів на базі Node.js, Express, MongoDB і Mongoose. Центральну роль виконує файл app.js, у якому підключаються middleware, шаблонізатор, маршрути, база даних і загальна серверна логіка. Водночас функціональність не зосереджена в одному місці: її поділено між каталогами models, routes, views, public і utils. Така структура забезпечує порядок у коді та зрозумілий розподіл відповідальності. Моделі описують сутності, маршрути обробляють запити, представлення формують інтерфейс, а допоміжні модулі виконують службові операції. Це ще не повна мікросервісна архітектура, але правильна основа для подальшого виділення окремих сервісів.

У підрозділі 3.2 розглянуто serverless-компоненти. У роботі прямо зазначено, що фактичні функції на зразок AWS Lambda або Google Cloud Functions у поточному репозиторії не реалізовано. Це коректна позиція, оскільки потенційні архітектурні рішення не підмінюються готовою реалізацією. Натомість визначено операції, які можуть бути винесені в serverless-модель у майбутньому: email-сповіщення, обробка контактних форм, генерація документів, обробка зображень, періодичні задачі, оновлення аналітики та реакція на зміну статусу автомобіля. Такий підхід відповідає

гібридній логіці роботи: основні бізнес-процеси залишаються в серверному контурі, а допоміжні події можуть виконуватися окремими функціями.

У підрозділах 3.3-3.5 проаналізовано взаємодію компонентів, безпеку, тестування та результати. Система працює за зрозумілою схемою: браузер надсилає запит, маршрут обробляє його, Mongoose-модель звертається до MongoDB, після чого сторінка формується через Handlebars. Перевірка охоплює маршрути, моделі, адміністративний інтерфейс, клієнтський каталог, звернення, бронювання та сервісні записи. Разом із тим для промислової експлуатації потрібні додаткові кроки: автентифікація, авторизація за ролями, захист адміністративних маршрутів, автоматизовані тести, контейнеризація, CI/CD, моніторинг, резервне копіювання і фактична реалізація serverless-компонентів. Отже, третій розділ підтверджує практичну здійсненність рішення та показує реальну основу для його подальшого розвитку.

ВИСНОВКИ

У кваліфікаційній роботі вирішено завдання розроблення та обґрунтування архітектурної моделі цифрового сервісу автоматизації обліку продажу автомобілів із використанням мікросервісного та serverless-підходів. У процесі виконання роботи було розглянуто теоретичні основи сучасних архітектур програмного забезпечення, визначено вимоги до системи, спроєктовано її структурну модель, проаналізовано практичну реалізацію програмного рішення та оцінено ефективність запропонованого підходу.

Проаналізовано сучасний стан автоматизації бізнес-процесів у сфері продажу автомобілів. Встановлено, що традиційні способи ведення обліку, зокрема паперові журнали, окремі електронні таблиці та застарілі монолітні системи, не забезпечують належного рівня гнучкості, прозорості та масштабованості. Такі підходи ускладнюють контроль за актуальністю даних, створюють ризик дублювання інформації, знижують швидкість обробки клієнтських звернень і не дають змоги ефективно керувати фінансовими та сервісними процесами. Особливо це проявляється в умовах, коли автосалон працює з різними категоріями транспортних засобів, клієнтськими заявками, сервісним обслуговуванням і адміністративним керуванням.

Визначено, що цифровий сервіс автоматизації обліку продажу автомобілів повинен підтримувати не лише базове збереження інформації про транспортні засоби, а й повний цикл роботи з даними. До такого циклу належать ведення каталогу автомобілів, розподіл транспортних засобів за категоріями, фільтрація моделей за параметрами, приймання клієнтських звернень, адміністрування записів, облік сервісного обслуговування, робота із зображеннями та формування інформаційної основи для подальших продажів. Тому система має бути не просто вебкаталогом, а керованим цифровим середовищем для підтримки процесів автосалону.

Досліджено поняття та роль архітектури програмного забезпечення як основи побудови інформаційної системи. Встановлено, що архітектура

визначає не тільки структуру програмного коду, а й принципи взаємодії компонентів, організацію роботи з даними, порядок розгортання, підтримку безпеки, масштабування та подальше супроводження системи. Для сервісу продажу автомобілів це має особливе значення, оскільки помилки в архітектурних рішеннях можуть призвести до неузгодженості даних, складності оновлення системи, зниження продуктивності та ускладнення підтримки.

У роботі розглянуто мікросервісний підхід і встановлено, що його основна перевага полягає у поділі системи на окремі логічні компоненти з власною зоною відповідальності. Для предметної області продажу автомобілів такими компонентами можуть бути каталог автомобілів, адміністративний модуль, клієнтський контур, сервісне обслуговування, модуль сповіщень і фінансово-договірна частина. Такий поділ спрощує розуміння структури системи, знижує залежність між частинами програмного рішення та створює основу для незалежного масштабування окремих модулів.

Окрему увагу приділено serverless-архітектурі. У результаті аналізу визначено, що цей підхід є доцільним не для всієї системи загалом, а для окремих подієвих операцій, які мають короткий життєвий цикл і не потребують постійного виконання серверного процесу. До таких операцій у межах досліджуваної системи можна віднести обробку контактних форм, надсилання електронних повідомлень, фільтрацію каталогів, завантаження зображень, генерацію документів або виконання окремих допоміжних задач. Саме такі компоненти можуть бути винесені в serverless-функції на наступних етапах розвитку системи.

Проведено порівняння мікросервісного та serverless-підходів. Встановлено, що мікросервісна архітектура краще підходить для основної бізнес-логіки, яка потребує стабільної роботи, збереження стану, контролю даних і тривалої взаємодії з користувачем. Serverless-підхід, навпаки, є ефективним для допоміжних, асинхронних і подієвих процесів. У зв’язку з цим обґрунтовано доцільність використання гібридної архітектурної моделі, у якій

основні функціональні контури реалізуються як сервісні модулі, а окремі події можуть оброблятися безсерверними функціями.

У процесі проєктування визначено функціональні та нефункціональні вимоги до системи. До функціональних вимог віднесено перегляд і фільтрацію каталогів автомобілів, роботу з електричними та бензиновими моделями, адміністрування записів, облік клієнтських звернень, керування сервісними заявками та підтримку службового доступу. До нефункціональних вимог віднесено продуктивність, надійність, безпеку, масштабованість, зручність використання, підтримуваність і готовність до подальшого розвитку. Саме нефункціональні вимоги визначають практичну якість системи, оскільки навіть функціонально повний сервіс не може вважатися готовим до реального використання без належного рівня безпеки, стабільності й контролю даних.

Розроблено концепцію архітектурної моделі цифрового сервісу. Її основу становить модульна серверна реалізація на базі Node.js та Express.js із використанням MongoDB як бази даних і Mongoose для опису моделей. Структура системи побудована таким чином, що окремі маршрути, моделі та шаблони відповідають за різні частини предметної області. Це дозволяє розглядати поточне рішення як модульний моноліт із чіткими ознаками сервісної декомпозиції. Такий підхід є доцільним для прототипу, оскільки не ускладнює розробку надмірною інфраструктурою, але водночас залишає можливість подальшого переходу до повноцінної мікросервісної архітектури.

У практичній частині досліджено програмне рішення Autorizz, реалізоване з використанням Node.js, Express.js, MongoDB, Mongoose та Handlebars. У структурі системи виокремлено адміністративний контур, каталог електромобілів, каталог автомобілів з двигуном внутрішнього згоряння, клієнтську взаємодію, сервісне обслуговування та модуль email-сповіщень. Наявність окремих маршрутів для /admin, /electric і /gas підтверджує логічне розділення функціональних зон. Це не є повноцінними незалежними мікросервісами у строгому технічному значенні, однак така структура є правильною основою для майбутньої декомпозиції.

Досліджено структуру моделей даних і встановлено, що система використовує окремі Mongoose-схеми для різних сутностей. Зокрема, моделі електромобілів і бензинових автомобілів мають власні набори атрибутів, що відповідають специфіці транспортних засобів. Такий підхід дозволяє уникнути надмірного узагальнення даних і забезпечує точніший опис предметної області. Крім того, використання Mongoose дає змогу задавати обов'язкові поля та базові правила збереження документів у MongoDB.

Розглянуто реалізацію адміністративної частини системи. Встановлено, що вона забезпечує вхід адміністратора, доступ до панелі керування, роботу з моделями автомобілів, клієнтськими записами, сервісними заявками та завантаженням зображень. Адміністративний модуль є важливою частиною системи, оскільки саме через нього працівник автосалону може керувати наповненням каталогу та контролювати службові процеси. Водночас у процесі аналізу виявлено, що механізм авторизації реалізований спрощено і потребує подальшого вдосконалення.

Виявлено компоненти, які можуть бути винесені в serverless-модель. Найбільш придатними до такого переходу є обробка контактної форми, надсилання електронних повідомлень, фільтрація каталогів, завантаження зображень і допоміжні подієві операції. Ці функції мають обмежений обсяг, чітко визначені вхідні та вихідні дані й не потребують постійного збереження стану. Тому їх винесення у serverless-функції може зменшити навантаження на основний застосунок і підвищити гнучкість системи.

Проведено тестування основних сценаріїв роботи системи. Перевірено відкриття каталогів електричних і бензинових автомобілів, фільтрацію моделей, перехід до сторінок бронювання, роботу адміністративного входу, перегляд сервісних записів, облік клієнтських звернень і функціонування окремих адміністративних сторінок. Результати тестування підтвердили, що система виконує ключові функції, передбачені предметною областю, та може розглядатися як працездатний прототип цифрового сервісу для автосалону.

Оцінено практичну ефективність реалізованого рішення. Встановлено, що цифровий підхід має суттєві переваги порівняно з ручним веденням обліку або використанням розрізнених таблиць. Система централізує інформацію про автомобілі, пришвидшує пошук потрібної моделі, дозволяє накопичувати клієнтські звернення, підтримує післяпродажний сервіс і забезпечує адміністративне керування контентом. Це підвищує прозорість роботи автосалону та зменшує ймовірність втрати або дублювання інформації.

Разом із позитивними результатами визначено технічні обмеження поточної реалізації. Найважливішими серед них є зберігання пароля у відкритому вигляді, відсутність повноцінного сесійного контролю або JWT-автентифікації, недостатня валідація вхідних даних, відсутність обмежень для завантажуваних файлів, локальне зберігання зображень і відсутність автоматизованого тестового контуру. Ці недоліки не заперечують працездатність прототипу, але свідчать про необхідність додаткового інженерного доопрацювання перед використанням системи в реальному середовищі.

Запропоновано напрями подальшого розвитку програмного рішення. Насамперед доцільно впровадити хешування паролів за допомогою bcrypt, реалізувати JWT або express-session для контролю доступу, додати валідацію даних за допомогою Joi або Zod, посилити захист HTTP-заголовків через Helmet, запровадити rate limiting, перенести зображення до хмарного сховища, налаштувати автоматизоване тестування та контейнеризувати систему за допомогою Docker. У перспективі система може бути переведена на API-first підхід із поступовим виділенням окремих доменів у самостійні мікросервіси.

Поставлену мету роботи досягнуто. Розроблено й обґрунтовано архітектурну модель цифрового сервісу автоматизації обліку продажу автомобілів, яка поєднує модульну серверну реалізацію, принципи мікросервісного підходу та можливість подальшого використання serverless-компонентів. Практична реалізація підтвердила, що запропонована модель придатна для автоматизації базових процесів автосалону.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Akkus I. E. et al. SAND: Towards high performance serverless computing // Proceedings of the 2018 USENIX Annual Technical Conference. USENIX Association, 2020. URL: https://www.usenix.org/conference/atc18/presentation/akkus .

2. Alam M. M. et al. Unveiling Serverless: A Systematic Review of Software Architecture and Developer Experience // IEEE, 2024. URL: https://ieeexplore.ieee.org/document/10895318 .

3. Anasuri S. Serverless Architecture Patterns: Deployment, Cost, and Latency Analysis // American International Journal of Computer Science and Technology, 2025. URL: https://aijcst.org/index.php/aijcst/article/view/104 .

4. Bhasin S. Ведучи майбутнє: Чому автомобільні компанії повинні... // LinkedIn, 2024. URL: https://ua.linkedin.com/pulse/driving-future-why-automotive-companies-must-embrace-strategy-bhasin-3ubac .

5. Buyya R. et al. A manifesto for future generation cloud computing // ACM Computing Surveys, 2020.

6. Cerny T. et al. Microservice Architecture Reconstruction and Visualization Techniques: A Review // arXiv:2207.02988, 2022. URL: https://arxiv.org/abs/2207.02988 .

7. Duessmann G., Fiorese A. Evaluating Serverless Function Deployment Models on AWS Lambda // ICEIS 2025. URL: https://www.scitepress.org/Papers/2025/132795/132795.pdf .

8. Era C. A. A. A Comprehensive Study of Microservices Architecture in Comparison with SOA and Monolithic Models // IEEE, 2025. URL: https://ieeexplore.ieee.org/document/10959671 .

9. Garofolo E. Practical Microservices: Build Event-Driven Architectures with Event Sourcing and CQRS. Pragmatic Bookshelf, 2020. URL: https://pragprog.com/titles/egmicro/practical-microservices/ .

10. Gilbert J., Lavi M. Software Architecture Patterns for Serverless Systems (2nd ed.). Packt Publishing, 2025. URL: https://www.packtpub.com/en-us/product/software-architecture-patterns-for-serverless-systems-9781803244433 .

11. Goyal M. Moving From Monolithic To Microservices Architecture for Multi-Agent Systems // arXiv:2505.07838, 2025. URL: https://arxiv.org/abs/2505.07838 .

12. Kumari A. Emergence of serverless computing in cloud: A state-of-the-art review of challenges and opportunities // Array, 2026. URL: https://www.sciencedirect.com/science/article/pii/S2666827026000277 .

13. Liu G. Microservices: architecture, container, and challenges // IEEE, 2020. URL: https://ieeexplore.ieee.org/document/9282637 .

14. Lyashenko V. I., Vishnevsky O. S. Цифрова модернізація економіки України (оновлене видання). К.: Інститут економіки промисловості НАН України, 2020–2024. URL: https://iie.org.ua/wp-content/uploads/monografiyi/2017/Lyashenko_Vishnevsky_2018.pdf .

15. Mateus-Coelho N. Security in Microservices Architectures // Procedia Computer Science, 2021. URL: https://www.sciencedirect.com/science/article/pii/S1877050921003719 .

16. Newman S. Building Microservices: Designing Fine-Grained Systems (2nd ed.). O’Reilly Media, 2021.

17. Newman S. Monolith to Microservices. O’Reilly Media, 2021 (оновлене видання). URL: https://samnewman.io/books/monolith-to-microservices/ .

18. Parikh A. Monolithic to Microservices Architecture - A Framework for Design and Implementation // IEEE, 2022. URL: https://ieeexplore.ieee.org/document/10072238 .

19. Pathak G. A Review of Cloud Microservices Architecture for Modern Applications // IEEE, 2023. URL: https://ieeexplore.ieee.org/document/10235199 .

20. Peczyński V. Patterns in Design of Microservices Architecture // WIPIEC Journal, 2025. URL: https://wipiec.digitalheritage.me/index.php/wipiecjournal/article/view/101 .

21. Richardson C. Microservices Patterns: With examples in Java (2nd ed.). Manning, 2025. URL: https://microservices.io/post/architecture/2025/06/26/announcing-meap-microservices-patterns-2nd-edition.html .

22. Saha T. Application Development Using Microservice Architecture. St. Cloud State University, 2022. URL: https://repository.stcloudstate.edu/cgi/viewcontent.cgi?article=1050&context=csit_etds .

23. Sawhney R., Chanumolu K. Beginning Azure Functions. Apress, 2023.

24. Скляренко О. В. Аналіз шляхів інтеграції цифрових технологій у виробничі процеси та організаційно-економічних моделей для забезпечення сталого розвитку автомобільної промисловості України // Journals KYMU, 2025. URL: https://journals.kymu.kyiv.ua/index.php/economy/article/view/255 .

25. Thatikonda V. K. Assessing the Impact of Microservices Architecture on Software Maintainability and Scalability,2023. URL: https://pdfs.semanticscholar.org/bdae/24d7245a5937fcf8fcf6cec25bd321e5f24a.pdf .

26. Velepucha V. A Survey on Microservices Architecture: Principles, Practices, and Challenges // IEEE, 2023. URL: https://ieeexplore.ieee.org/document/10220070 .

27. Vanisree M. Cloud Computing and Serverless Architectures Innovations and Applications // IEEE, 2025. URL: https://ieeexplore.ieee.org/document/11071764 .

28. Willard J. The Evolution and Future of Microservices Architecture with AI-Driven Enhancements // Digital Commons, Lindenwood University, 2025. URL: https://digitalcommons.lindenwood.edu/faculty-research-papers/718/ .

29. Weerasinghe L. Reference Architecture for Microservices with an Optimized Inter-Service Communication Strategy // IEEE, 2024. URL: https://ieeexplore.ieee.org/document/10550466 .

30. Зварич Р. Інноваційна трансформація автомобільної галузі в умовах глобального «зеленого» переходу // Visnyk of West Ukrainian National University, 2025. URL: https://visnykj.wunu.edu.ua/index.php/visnykj/article/view/1838 .

31. Alanezi K. Utilizing Microservices Architecture for Enhanced Service Provisioning at the Edge // IEEE, 2022. URL: https://ieeexplore.ieee.org/document/9864183 .

32. Chen C. et al. Design and Implementation of Software Architecture for Automotive Green Supply Chain Based on Microservices // IOP Conference Series, 2019 (оновленийконтекст 2020+). URL: https://iopscience.iop.org/article/10.1088/1742-6596/1314/1/012096 .

33. Daraujimba A. I. et al. Systematic Review of Serverless Architectures and Business Process Optimization // IRE Journals, 2025. URL: https://www.irejournals.com/formatedpaper/1708517.pdf .

34. Dhonde A. Futuristic Automotive Software Architecture: Microservices, Orchestration and Networked Hardware // IEEE SA, 2023. URL: https://standards.ieee.org/wp-content/uploads/2023/10/11-futuristic-aumotive.pdf .

35. GS Harsha. The Microservices Revolution in Software-Defined Vehicles (SDVs) // LinkedIn / Medium, 2025. URL: https://www.linkedin.com/pulse/microservices-revolution-software-defined-vehicles-sdvs-harsha-gs-5ycae .

36. Liu H. Research on the Application of Digital Technology in Automotive Software Update Management System // ACM, 2025. URL: https://dl.acm.org/doi/10.1145/3796731.3796869 .

37. MONO2REST: Identifying and Exposing Microservices // arXiv:2503.21522, 2025. URL: https://arxiv.org/abs/2503.21522 .

38. Petrasch R. et al. Model-based engineering for microservice architectures (оновленийконтекст). IEEE, 2017–2022.

39. Sofico. Survival of the fittest: how microservices are redefining automotive financing, 2024. URL: https://sofico.global/en/story/survival-of-the-fittest-how-microservices-are-redefining-automotive-financing/ .

40. Virtualization & Microservice Architecture for Software-Defined Vehicles // arXiv:2412.09995, 2024. URL: https://arxiv.org/abs/2412.09995 .

41. Waseem M. Microservices Architecture in DevOps // IEEE, 2017–2022+ (контекст).

42. Avangarde Software. Microservices architecture and design for Automotive. URL: https://avangarde-software.com/cases/microservices-architecture-and-design-for-automotive/ .

43. LTTS. Driving Innovation: How Microservices and DevOps Adoption is Shaping the Future of Automotive Software, 2025. URL: https://www.ltts.com/blog/automotive-software .

44. Development Frameworks for Microservice-based Applications // arXiv:2203.07267, 2022. URL: https://arxiv.org/abs/2203.07267 .

45. Case study on data communication in microservice architecture // ACM, 2020+. URL: https://dl.acm.org/doi/10.1145/3338840.3355659 .

46. You don't need a Microservices Architecture (yet) // ACM, 2021. URL: https://dl.acm.org/doi/10.1145/3501774.3501780 .

47. Microservice Architecture Patterns for Enterprise Applications // ACM, 2023. URL: https://dl.acm.org/doi/10.5555/3631672.3631698 .

48. Transforming Monolithic Systems to a Microservices Architecture // ACM, 2023. URL: https://dl.acm.org/doi/10.1145/3573074.3573091 .

49. Serverless Architecture and Its Current State of the Art: A Systematic Literature Review // Preprints.org, 2025. URL: https://www.preprints.org/manuscript/202512.0219/v1 .

50. Microsoft. Рішення корпорації Майкрософт для автомобільної галузі. URL: https://www.microsoft.com/uk-ua/ai/mobility/automotive .

51. Цифрова трансформація: виклики та стратегії: матеріали Міжнародної науково-практичної конференції. Луцьк: ЛНТУ, 2025. URL: https://lib.lntu.edu.ua/sites/default/files/2025-06/%D0%97%D0%B1%D1%96%D1%80%D0%BD%D0%B8%D0%BA%D0%9B%D0%9D%D0%A2%D0%A3_2.pdf .

52. Kriskun, I., & Bondarchuk, A. (2026). Оцінювання ефективності управління проєктами в галузі інформаційних технологій. Європейський науковий журнал Економічних та Фінансових інновацій, 1(19), 264-274.

53. Node.js Microservices with MongoDB, Docker & RabbitMQ (tutorial examples). Various sources, 2024–2025 (e.g., YouTube/Dev.to tutorials on practical implementation).

54. Mastering Microservices: A Hands-On Tutorial with Node.js, RabbitMQ, Nginx, and Docker // DEV Community, 2024. URL: https://dev.to/davydocsurg/mastering-microservices-a-hands-on-tutorial-with-nodejs-rabbitmq-nginx-and-docker-m4f .

55. Abramov, V., Astafieva, M., Boiko, M., Bodnenko, D., Bushma, A., Vember, V., ... & Yaskevych, V. (2021). Theoretical and practical aspects of the use of mathematical methods and information technology in education and science.

56. Машкіна, І. В., Абрамов, В. О., Носенко, Т. І., Іваніченко, Є. В., & Яскевич, В. О. (2024). Навчально-методичний посібник до виконання і захисту кваліфікаційної роботи бакалавра спеціальностей 122 Комп’ютерні науки для здобувачів вищої освіти денної форми навчання.

Додаток А. Лістинг коду основної архітектури застосунку (app.js та монтування мікросервісів)

```
const express = require('express');
const mongoose = require('mongoose');
const electricRouter = require('./routes/electric_index');
const gasRouter = require('./routes/gas_index');
const adminRouter = require('./routes/admin');

const app = express();

// Підключеннядо MongoDB
const db = async () => {
    try {
        await mongoose.connect('mongodb://localhost:27017/autorizz', {
            useNewUrlParser: true,
            useUnifiedTopology: true
        });
        console.log("MongoDB connected");
    } catch (err) {
        console.log("MongoDB Error", err);
        process.exit(1);
    }
};
db();

// Налаштування view engine (Handlebars)
app.engine('.hbs', exphbs({ defaultLayout: 'layout', extname: '.hbs' }));
app.set('view engine', '.hbs');

app.use(express.json());
app.use(express.urlencoded({ extended: false }));
app.use(express.static(path.join(__dirname, 'public')));

// Монтуваннялогічнихмікросервісів (роутерів)
app.use('/admin', adminRouter);
app.use('/electric', electricRouter);
app.use('/gas', gasRouter);

// Потенційний serverless-кандидат – обробка контактної форми
app.post('/customer', async (req, res) => {
    const user = new UserModel({
        name: req.body.username,
        email: req.body.useremail,
        phone: req.body.userphone
    });
    const user_res = await user.save();
    console.log(user_res);
});
module.exports = app;
```

Додаток Б. Моделі даних (Mongoose схеми для ключових сутностей)

ElectricModel.js (приклад моделі електромобілів):

```
const mongoose = require("mongoose");

const electricModelSchema = new mongoose.Schema({
    imagePath: { type: String, required: true },
    title: { type: String, required: true },
    t1: { type: String, required: true },
    t2: { type: String, required: true },
    year: { type: Number, required: true },
    price: { type: Number, required: true },
    priceStr: { type: String, required: true },
    topspeed: { type: String, required: true },
    time60: { type: String, required: true },
    range: { type: String, required: true },
    colour: { type: String, required: true },
    interior: { type: String, required: true },
    wheel: { type: String, required: true },
    description: { type: String, required: true },
    safety: { type: String, required: true },
    rangedesc: { type: String, required: true }
});

module.exports = mongoose.model("electricmodel",
electricModelSchema, "electricmodel");
```

Додаток В. Приклади потенційних serverless-компонентів та безпекових рекомендацій

Фрагмент обробки контактної форми (app.js) serverless-функція:

JavaScript

```
app.post('/customer', async (req, res) => {
    const user = new UserModel({
        name: req.body.username,
        email: req.body.useremail,
        phone: req.body.userphone
    });
    const user_res = await user.save();
    console.log(user_res);
});
2. Утилітанадсилання email (utils/mailer.js) -
асинхроннаподієваоперація:
JavaScript

const nodemailer = require("nodemailer");

async function sendEmail(clientEmail) {
    const htmlTemp = '<h4>Dear Customer, Vehicle service is now
completed...</h4>';
    const transporter = nodemailer.createTransport({
        service: 'gmail',
        auth: { user: 'xxxx@gmail.com', pass: 'xxxx' }
    });

    const mailOptions = {
        from: "xxxx@gmail.com",
        to: clientEmail,
        subject: "Autorizz Service Update",
        html: htmlTemp
    };
    try {
        await transporter.sendMail(mailOptions);
        return true;
    } catch (error) {
        console.log(error);
        return false;
    }
}

module.exports = sendEmail;
```

Додаток Г. Порівняльна таблиця

Технологія / Інструмент	Версія (рекомендована 2026)	Причина вибору (детально)	Альтернатива та чому не основна
Node.js	20.x або 22.x LTS	Асинхронність (non-blocking I/O), висока продуктивність при великій кількості одночасних запитів (перегляд каталогу, бронювання). Ефективно обробляє 200+ користувачів. Велика екосистема npm.	Bun / Deno - швидші, але менша зрілість екосистеми.
Express.js	4.x (або Fastify як апгрейд)	Простота та мінімалізм маршрутизації, швидке створення REST API (/api/cars, /api/contracts). Легко інтегрується з middleware (авторизація, валідація).	NestJS - для складної архітектури (модулі, DI). Fastify - вища швидкість.
MongoDB (Atlas або self-hosted)	7.x / 8.x	Гнучкість схеми: автомобілі мають різну комплектацію, фото, історію статусів. Швидке зберігання JSON-подібних документів. Горизонтальне масштабування. Відмінно підходить для automotive (реальні кейси CarGurus, connected vehicles).	PostgreSQL - якщо потрібні складні транзакції та joins (фінанси). Використовувати як гібрид.
Mongoose	8.x	ODM (Object Document Mapper) для Node.js: схеми, валідація, middleware, populate для зв'язків (Авто ↔ Клієнт ↔ Договір). Спрощує роботу з БД.	Prisma / Typegoose - сучасніші з TypeScript.
Handlebars (express-handlebars)	7.x	Простий шаблонізатор для серверного рендерингу (SSR). Легко створювати клієнтський інтерфейс і адміністративну панель без складного frontend-фреймворку. Підходить для MVP.	EJS (простіший), Pug (компактніший), Nunjucks (логіка). Для сучасності - міграція на React/Next.js.
TypeScript	5.x	Статична типізація → менше помилок у великих проєктах (моделі авто, контракти, фінанси). Код sharing між frontend/backend. Рекомендовано в усіх сучасних JS-проєктах.	Чистий JS - швидше на старті, але гірше для підтримки.
JWT + bcrypt / Passport	-	Автентифікація та авторизація (ролі: admin, manager, accountant). Захист /admin.	Auth0 / Firebase Auth - для швидкості, але дорожче.
Redis	7.x	Кешування (каталог авто, сесії), rate limiting, черги завдань (генерація PDF).	-
Docker + Docker Compose	-	Контейнеризація для легкого розгортання та розробки (Node + Mongo + Redis).	Kubernetes - для великого масштабу.