ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Метод синхронізації та підвищення ефективності взаємодії POS-систем із зовнішніми сервісами доставки на основі інтеграції систем обміну даними»

на здобуття освітнього ступеня магістра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Олександр ГОНГАЛО
(підпис)

Виконав: здобувач вищої освіти групи ПДМ-63
Олександр ГОНГАЛО

Керівник: Юрій ЗАДОНЦЕВ
канд техн. наук

Рецензент: _____
науковий ступінь, вчене звання *Ім'я, ПРІЗВИЩЕ*

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Магістр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ

Завідувач кафедри

Інженерії програмного забезпечення

_____ Ірина ЗАМРІЙ

«_____» _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Гонгалу Олександру Олександровичу

1. Тема кваліфікаційної роботи: «Метод синхронізації та підвищення ефективності взаємодії POS-систем із зовнішніми сервісами доставки на основі інтеграції систем обміну даними»

керівник кваліфікаційної роботи Юрій ЗАДОНЦЕВ, канд. техн. наук, доцент кафедри ІПЗ,

затверджені наказом Державного університету інформаційно-комунікаційних технологій від «30» жовтня 2025 р. № 467.

2. Строк подання кваліфікаційної роботи «19» грудня 2025 р.

3. Вихідні дані до кваліфікаційної роботи: науково-технічна література з

POS-систем, інтеграції та синхронізації даних; моделі інтеграції (API, middleware, event-driven, ESB, webhooks); формати обміну (JSON/XML/Protobuf); засоби безпеки (OAuth 2.0/JWT, TLS); черги/брокери повідомлень (Kafka/RabbitMQ або аналог); методики оцінювання продуктивності та надійності; тестовий стенд/прототип інтеграції; інструменти моніторингу та логування.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Провести аналіз предметної області та проблем синхронізації між POS-системами і сервісами доставки; порівняти існуючі підходи інтеграції та обміну даними (API-орієнтована, event-driven, ESB, webhooks тощо).
2. Розробити концептуальну модель інтеграції POS ↔ сервіс доставки на основі системи обміну даними (шина/брокер повідомлень, адаптери, уніфікація схем даних, трансформації).
3. Запропонувати метод синхронізації та підвищення ефективності взаємодії (алгоритми доставки подій, забезпечення консистентності, ідемпотентність, повторні спроби/backoff, дедуплікація, контроль версій/статусів, моніторинг).
4. Виконати експериментальну апробацію: реалізувати тестовий стенд/прототип і провести оцінювання ефективності (затримка, пропускна здатність, стабільність при збоях, робота при пікових навантаженнях) з порівнянням базового та запропонованого підходів.

5. Перелік ілюстративного матеріалу: *презентація*

- Схема учасників інтеграції (POS, сервіс доставки, інтеграційний шар, брокер повідомлень).
- Порівняльна характеристика моделей інтеграції (пряма API, middleware, hub/агрегатор, event-driven).
- Діаграма потоків даних: меню/ціни/залишки, замовлення, статуси доставки.
- Алгоритм синхронізації замовлення та статусів у подієвій моделі.
- Алгоритм обробки помилок (retry/backoff) та забезпечення ідемпотентності.
- Архітектура прототипу/тестового стенда (компоненти, черги/топіки, адаптери, сховище, логування/моніторинг).
- Діаграми результатів експериментів: latency/throughput/error-rate, порівняння «до/після», робота при піковому навантаженні.

6. Дата видачі завдання «31» жовтня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз наявної науково-технічної літератури щодо POS-систем, сервісів доставки та інтеграції систем обміну даними	31.10 – 07.11.2025	
2	Аналіз методів і технологій інтеграції POS із зовнішніми сервісами доставки (API, webhooks, middleware, event-driven, брокери повідомлень)	08.11 – 12.11.2025	
3	Аналіз проблем ефективності взаємодії та синхронізації (затримки, дублювання, узгодженість статусів, пікові навантаження, збої, безпека)	13.11 – 15.11.2025	
4	Розробка концептуальної моделі та постановка задачі підвищення ефективності інтеграції POS-систем і служб доставки	16.11 – 18.11.2025	
5	Розділ 2. Моделювання та обґрунтування методу підвищення ефективності інтеграції pos-систем і служб доставки	19.11 – 21.11.2025	
6	Розділ 3. Програмна реалізація методу та оцінка ефективності інтеграції POS-систем і служб доставки	22.11 – 04.12.2025	
7	Оформлення роботи: вступ, висновки, реферат, список джерел (ДСТУ 8302:2015), додатки; підготовка до нормконтролю	05.12 – 16.12.2025	
8	Розробка демонстраційних матеріалів: презентація, схеми/діаграми, блок-схеми алгоритмів, графіки результатів	17.12 – 19.12.2025	

Здобувач вищої освіти _____ Олександр ГОНГАЛО
(підпис)

Керівник
кваліфікаційної роботи _____ Юрій ЗАДОНЦЕВ
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 70 стор., 8 табл., 12 рис., 35 джерел.

Мета роботи – підвищення ефективності, надійності та стабільності взаємодії POS-систем із зовнішніми сервісами доставки за рахунок удосконалення методів синхронізації та інтеграції систем обміну даними.

Об’єкт дослідження – процес синхронізації та обміну даними між POS-системами та зовнішніми сервісами доставки у розподілених інформаційних системах.

Предмет дослідження – метод синхронізації та підвищення ефективності взаємодії POS-систем із зовнішніми сервісами доставки на основі інтеграції систем обміну даними.

У роботі використано методи системного та порівняльного аналізу, моделювання інтеграційних процесів, подієво-орієнтовані підходи до побудови розподілених систем, а також методи експериментальної оцінки продуктивності та надійності інформаційних систем.

Проведено аналіз сучасних методів синхронізації та обміну даними між інформаційними системами, зокрема підходів, орієнтованих на API, подієво-орієнтованих моделей, використання корпоративної шини даних, пакетної синхронізації та механізмів вебхуків. Виявлено основні обмеження традиційних рішень у контексті високонавантажених сценаріїв роботи POS-систем і служб доставки.

Розроблено та обґрунтовано метод синхронізації взаємодії POS-систем із зовнішніми сервісами доставки на основі інтеграції систем обміну даними з використанням подієво-орієнтованої архітектури, брокера повідомлень, механізмів повторних спроб і ідемпотентної обробки подій.

Проведено експериментальні дослідження для оцінювання ефективності запропонованого методу за показниками затримки обробки замовлень, надійності передачі даних та відмовостійкості системи, що підтвердило доцільність його застосування у реальних POS-рішеннях.

КЛЮЧОВІ СЛОВА: POS-СИСТЕМИ, СИНХРОНІЗАЦІЯ ДАНИХ, ІНТЕГРАЦІЯ СИСТЕМ, EVENT-DRIVEN АРХІТЕКТУРА, БРОКЕР ПОВІДОМЛЕНЬ, API, СЛУЖБИ ДОСТАВКИ, ВІДМОВОСТІЙКІСТЬ, РОЗПОДІЛЕНІ СИСТЕМИ.

ABSTRACT

Textual part of the qualification work for obtaining the Master's degree: 70 pages, 8 tables, 12 figures, 35 references.

The purpose of the work is to improve the efficiency, reliability, and stability of interaction between POS systems and external delivery services by enhancing data synchronization methods and data exchange system integration.

The object of the research is the process of data synchronization and exchange between POS systems and external delivery services in distributed information systems.

The subject of the research is a method for synchronizing and improving the efficiency of interaction between POS systems and external delivery services based on the integration of data exchange systems.

The work employs methods of system and comparative analysis, modeling of integration processes, event-driven approaches to the design of distributed systems, as well as methods for experimental evaluation of performance and reliability of information systems.

An analysis of modern methods of data synchronization and exchange between information systems has been conducted, including API-oriented approaches, event-driven models, the use of an enterprise service bus, batch synchronization, and webhook mechanisms. The main limitations of traditional solutions in the context of high-load operation scenarios of POS systems and delivery services have been identified.

A method for synchronizing the interaction between POS systems and external delivery services based on the integration of data exchange systems has been developed and substantiated.

The method utilizes an event-driven architecture, a message broker, retry mechanisms, and idempotent event processing.

Experimental studies were conducted to evaluate the effectiveness of the proposed method in terms of order processing latency, data transmission reliability, and system fault tolerance, which confirmed the feasibility of its application in real POS solutions.

KEYWORDS: POS SYSTEMS, DATA SYNCHRONIZATION, SYSTEM INTEGRATION, EVENT-DRIVEN ARCHITECTURE, MESSAGE BROKER, API, DELIVERY SERVICES, FAULT TOLERANCE, DISTRIBUTED SYSTEMS.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API – прикладний програмний інтерфейс (Application Programming Interface)

ACK – підтвердження отримання повідомлення (Acknowledgement)

Batch processing – пакетна обробка даних

CRM – система управління взаємовідносинами з клієнтами (Customer Relationship Management)

EDA – подієво-орієнтована архітектура (Event-Driven Architecture)

ERP – система планування ресурсів підприємства (Enterprise Resource Planning)

ESB – корпоративна шина даних (Enterprise Service Bus)

HTTP – протокол передачі гіпертексту (HyperText Transfer Protocol)

JSON – формат обміну структурованими даними (JavaScript Object Notation)

Latency – затримка обробки або передачі даних

POS – система обліку та обробки продажів (Point of Sale)

Pub/Sub – модель публікації та підписки на події (Publish/Subscribe)

QoS – якість обслуговування (Quality of Service)

REST – архітектурний стиль взаємодії веб-сервісів (Representational State Transfer)

Retry – механізм повторної спроби виконання операції

SLA – угода про рівень обслуговування (Service Level Agreement)

SLO – цільовий рівень обслуговування (Service Level Objective)

SOA – сервіс-орієнтована архітектура (Service-Oriented Architecture)

Throughput – пропускна здатність системи

Webhook – механізм асинхронного зворотного виклику через HTTP

XML – мова розмітки для обміну структурованими даними (eXtensible Markup Language)

ЗМІСТ

ВСТУП

Сучасний бізнес не може функціонувати так, як ми його знаємо, якщо цифрові системи, такі як цифрові технології, не підтримують процеси обміну даними на високій швидкості та з точністю. Наріжним каменем такого цифрового екосистеми є POS (Point of Sale) системи — це багатокомпонентні системи, які інтегрують продажі та облік даних, управління запасами, аналітичні послуги та управління взаємодією з клієнтами.

У торгівлі та громадському харчуванні POS система стала «центральним вузлом» для операцій, мозком для більшості процесів від продажу до інвентаризації та фінансових транзакцій. За останні кілька років бізнес-модель у цих сферах значно трансформувалася під впливом онлайн-сервісів доставки (Glovo, Bolt Food, Raketa, Uber Eats та інші). Ці платформи все більше стають основним способом спілкування між закладами та споживачами.

Водночас існує безліч операційних розривів між ними та POS системами: замовлення не оновлюються синхронно, є затримки між статусами, а дані про наявність товарів або страв часто не збігаються. Результатом є помилки в замовленнях, втрата часу персоналу, зниження рівня обслуговування та втрата грошей.

Це актуально в контексті реального часу, оскільки в момент цифрової трансформації саме інтеграція та синхронізація інформаційних систем визначатимуть, чи буде підприємство конкурентоспроможним у цій сфері. І хоча великі корпорації мають власні рішення для автоматизації цих процесів, малі та середні підприємства зазвичай покладаються на POS системи без гнучкого рішення для взаємодії з зовнішніми системами доставки. Через це дані дублюються, запити обробляються вручну або з затримкою, що значно знижує ефективність всієї системи.

З огляду на динамічний розвиток цифрової економіки, інтеграція розподілених інформаційних систем стала не лише технічною вимогою, а й конкурентною перевагою. Для бізнесів, що працюють у сфері торгівлі чи

громадського харчування, ключовим фактором стало забезпечення наскрізного й безперебійного обміну інформацією між внутрішніми системами обліку та зовнішніми сервісами, зокрема платформами доставки. Успішна синхронізація таких систем передбачає подолання як технічних, так і організаційних бар'єрів: різні формати даних, відсутність єдиного протоколу обміну, нестабільність зовнішніх API та обмежена керованість даними в реальному часі. Усі ці фактори знижують ефективність бізнес-процесів і призводять до збоїв, які прямо впливають на клієнтський досвід. Саме тому зростає потреба в створенні універсальних підходів до побудови гнучких, надійних і масштабованих інтеграційних рішень, які дозволяють узгоджувати взаємодію між POS-системами та службами доставки незалежно від конкретних технічних реалізацій. З огляду на це, у даній роботі запропоновано нову архітектурну модель інтеграції, яка враховує реальні виклики та дозволяє досягти високої якості обміну в умовах децентралізованого цифрового середовища.

Особливу складність становить забезпечення консистентності даних між розподіленими системами, що працюють у різних технологічних та часових доменах. Наприклад, зміна статусу замовлення в POS-системі має миттєво відображатися у зовнішній платформі доставки, і навпаки — будь-яке оновлення або скасування з боку клієнта має бути оперативно враховане у внутрішніх бізнес-процесах закладу. У контексті високої інтенсивності транзакцій, типових для годин пік, навіть незначна затримка або втрата події може призвести до критичних наслідків: дублювання замовлень, порушення логістики або незадоволення клієнтів.

Метою роботи є дослідження існуючих методів і розробка способу інтеграції та підвищення ефективності взаємодії між POS-системами та зовнішніми службами доставки на основі інтеграції систем обміну даними.

У межах поставленої мети передбачено такі завдання дослідження:

– проаналізувати архітектуру та функціональні можливості сучасних POS-систем;

– пояснити особливості їх інтеграції із зовнішніми сервісами доставки;

– визначити існуючі методи синхронізації даних і виявити їх переваги та

недоліки;

– сформулювати вимоги до ефективного обміну даними між системами;

– запропонувати узагальнену модель або процедуру, що сприятиме підвищенню ефективності взаємодії.

Об’єкт дослідження — інтеграційні системи між POS та зовнішніми службами доставки.

Предмет дослідження — техніки, моделі та технології синхронізації даних у таких системах.

Методи дослідження. У роботі застосовано порівняльний аналіз наукових джерел, системно-аналітичний і структурно-логічний підходи, а також аналіз архітектур сучасних інформаційних систем. Для оцінки ефективності запропонованого рішення використовуються методи моделювання й верифікації алгоритмів обміну даними.

Практична цінність отриманих результатів полягає у можливості впровадження розробленого процесу для підвищення швидкості та стабільності інтеграції POS-систем із сервісами доставки. Запропонований метод може бути реалізований як у новому програмному забезпеченні, так і в існуючих рішеннях шляхом оптимізації API-інтерфейсів або створення проміжного шару обміну даними. Це дає змогу прискорити обробку замовлень, зменшити втрати від десинхронізації й гарантувати актуальність інформації в реальному часі.

Наукова новизна роботи полягає у створенні структурованого підходу до побудови інтеграційної моделі на основі поєднання POS-системи та зовнішнього сервісу доставки, а також у розробленні методу синхронізації, який може бути застосований у межах єдиної системи обміну даними.

Структура роботи. Магістерська кваліфікаційна робота складається зі вступу, трьох розділів, висновків і списку використаних джерел. У першому розділі подано аналітичний огляд сучасних POS-систем, методів інтеграції та особливостей взаємодії з платформами доставки. Другий розділ присвячено обґрунтуванню вибору методів синхронізації, моделюванню архітектури інтеграційного шару та побудові алгоритмів обробки подій. У третьому розділі

наведено програмну реалізацію розробленого методу, результати експериментального дослідження та оцінку його ефективності.

1 АНАЛІЗ POS-СИСТЕМ ТА СУЧАСНИХ ПІДХОДІВ ДО ІНТЕГРАЦІЇ Й СИНХРОНІЗАЦІЇ ЗІ СЛУЖБАМИ ДОСТАВКИ

1.1 Сутність та архітектура POS-систем

Сучасні POS-системи (Point of Sale) загалом не є просто електронними касовими апаратами, а мають повне рішення для інтеграції даних та інформації, що пропонує автоматизовану торгівлю, систему фінансових розрахунків та автоматизацію процесу продажу і контролю запасів. Сьогоднішні POS-системи можна назвати чим завгодно, від онлайн-систем оформлення замовлень без прямого введення чи оплати до касових апаратів, обслуговування клієнтів та автоматизації POS-систем. Дані клієнтів, включаючи деталі, збережені у формі транзакцій, платіжних систем та платіжних карток, є центральною частиною бізнес-інфраструктури роздрібних підприємств, ресторанів та сектору послуг. Звичайна POS-система створюється з трьох рівнів у своїй архітектурі, таких як дизайн POS-системи:

1. Клієнтський рівень, що включає інтерфейс касира, інтерфейси введення-виведення та периферійні пристрої (сканери, принтери чеків, платіжні термінали тощо);
2. Рівень бізнес-логіки, який підтримує обробку транзакцій, наявність товарів, перевірку знижок, обчислення податків;
3. Серверний рівень (переважно для зберігання даних у базі даних, взаємодії з бухгалтерськими модулями, CRM-системами та іншими платформами, обслуговування додатків доставки та аналізу). Архітектурно сучасні POS-рішення базуються на хмарних або мікросервісних платформах, що робить ці рішення більш придатними для великих організацій, пропонуючи масштабованість та постійну ефективність. На відміну від попередніх систем "все в одному", нова система може бути оновлена без зупинки

робочого процесу, а інтеграція з зовнішніми API спрощена завдяки цим новим технологіям.

Основні вимоги до цих систем такі:

- Швидкість обробки транзакцій (у межах сотих часток секунди на транзакцію);
- Цілісність даних (атомарність, узгодженість, ізоляція, довговічність — властивості ACID);
- Надійність та відмовостійкість (резервне копіювання, дублювання серверів);
- Безпека особистої та фінансової інформації;
- Гнучкість рівня інтеграції для підключення зовнішніх сервісів. POS-системи еволюціонують від локальних програм касових апаратів до розподілених веб-сервісів, здатних обробляти сотні замовлень за секунду, одночасно синхронізуючись із зовнішніми джерелами даних через API-інтерфейси або черги повідомлень.

З часом POS-системи еволюціонували від звичайних локальних касових програм до розподілених веб-платформ, здатних обробляти сотні замовлень за секунду. Наприклад, сучасний ресторан може одночасно приймати замовлення через мобільний додаток, сайт і термінал у залі — а POS-система синхронізує всі дані в режимі реального часу.

У результаті POS-система сьогодні — це не просто технологічний інструмент, а «мозок» торгового бізнесу. Вона забезпечує безперервність операцій, точність аналітики, комфорт клієнтів і стабільність доходів підприємства.

1.2 Особливості інтеграції POS-систем з зовнішніми службами доставки

Зростаюча популярність онлайн-служб доставки — таких як Glovo, Bolt Food, Raketa, Wolt та інші — змінила підхід до роботи сучасних закладів громадського харчування та роздрібної торгівлі. Якщо раніше прийом замовлень і оновлення меню здійснювались вручну, то сьогодні ці процеси автоматизовані завдяки інтеграції між внутрішньою POS-системою та API зовнішніх платформ доставки.

Така інтеграція дозволяє автоматично приймати замовлення безпосередньо на інтерфейсі POS, синхронізувати меню, ціни, наявність товарів і відображати статус доставки в режимі реального часу. Усі завершені замовлення відразу потрапляють у фінансові звіти, що усуває потребу у ручному введенні даних та зменшує кількість помилок.

У практиці використовується кілька основних моделей інтеграції:

1. Пряма інтеграція (API) — POS підключається безпосередньо до REST або GraphQL інтерфейсу служби доставки. Головна перевага — миттєва передача даних і відсутність затримки. Недолік — необхідність адаптації під кожну платформу окремо, що збільшує час розробки.
2. Інтеграція через проміжний сервер (middleware) — між системами створюється додатковий шар обробки запитів. Це дозволяє стандартизувати обмін даними та використовувати універсальні протоколи для кількох сервісів одночасно.
3. Інтеграція через агрегатор (hub) — використання спеціальної проміжної платформи, яка вже має налаштовані зв’язки з популярними службами доставки (наприклад, Delivery Hero, Takeaway API, Uber Eats API). У цьому випадку POS-система підключається лише до одного агрегатора, а він — до десятків партнерських сервісів. Такий підхід значно скорочує час впровадження.

Однією з ключових технічних складностей інтеграції є різниця у форматах даних — JSON, XML або CSV, а також у методах авторизації (OAuth2, токени доступу, API-ключі тощо). Через це виникає потреба у механізмах перевірки сумісності, фільтрації й трансформації даних між системами.

Крім того, платформи доставки працюють у асинхронному режимі — статуси замовлень можуть оновлюватися з різною швидкістю або надходити із затримкою. Для уникнення втрати інформації у випадку збою використовують черги повідомлень (Kafka, RabbitMQ), які забезпечують гарантовану доставку даних між серверами.

Інтеграція також потребує уваги до питань безпеки — зокрема, шифрування переданих даних, перевірки цифрових підписів запитів і обмеження доступу до внутрішніх систем. Особливо це важливо, коли через POS проходять персональні дані клієнтів або платіжна інформація.

Водночас правильно налагоджена інтеграція створює нові можливості для бізнесу. Власник ресторану або магазину бачить усі замовлення з різних платформ у єдиному вікні, може аналізувати популярність страв чи товарів, оптимізувати запаси й швидше реагувати на попит. У підсумку це підвищує ефективність управління, зменшує витрати часу персоналу та покращує якість обслуговування клієнтів.

1.3 Сучасні методи синхронізації та обміну даними

Синхронізація даних у розподілених інформаційних системах може відбуватися як у синхронному, так і в асинхронному режимі, що визначає характер взаємодії між компонентами та швидкість реакції на зміни. Синхронний обмін передбачає негайну відповідь системи на запит, однак створює залежність між сервісами та підвищує ризик затримок у разі нестабільності мережі. Асинхронний підхід, навпаки, дозволяє обробляти події незалежно від доступності окремих компонентів, використовуючи черги повідомлень, механізми підтвердження доставки та часові мітки, які забезпечують правильний порядок оновлень. Важливим аспектом є також вибір між повними пакетами даних і дельта-оновленнями. Передача лише змінених фрагментів зменшує навантаження на канали зв’язку та скорочує час обробки, що критично у системах із високою частотою транзакцій.

Взаємодія POS-систем із сервісами доставки вимагає дотримання підвищених вимог до швидкодії, узгодженості та передбачуваності даних. Зміна статусу замовлення або наявності товару має миттєво відображатися у всіх задіяних системах, оскільки затримка навіть у кілька секунд може призвести до дублювання операцій або появи некоректних записів. Ідемпотентність операцій

відіграє ключову роль у запобіганні помилкам при повторних запитах, а стійкість до нестабільності API зовнішніх партнерів визначає загальну надійність інтеграційної архітектури. Саме ці вимоги зумовлюють необхідність переходу від найпростішої прямої синхронізації через API до більш розвинених моделей, таких як подієво-орієнтована взаємодія, використання корпоративної шини даних або спеціалізованих адаптерів. У той час як batch-процеси залишаються спадковою технологією, вони значно поступаються сучасним методам у швидкості та актуальності даних, тоді як розвинуті інструменти моніторингу стають обов'язковою умовою підтримання цілісності та стабільності обміну в реальному часі. Це створює логічне підґрунтя для подальшого детального розгляду окремих методів синхронізації у підпунктах 1.3.1–1.3.6.

1.3.1 Синхронізація, орієнтована на API

Цей підхід є найпоширенішим у сучасних рішеннях. Він базується на використанні REST або GraphQL API, через які POS-система надсилає запити до серверів служб доставки, отримуючи у відповідь структуровані дані у форматах JSON, XML або YAML. Така модель проста, зрозуміла та добре підходить для невеликих і середніх підприємств, де обсяги замовлень не перевищують декілька сотень на день. Проте API-синхронізація має і свої недоліки. Через велику кількість одночасних запитів у пікові години система може відчувати затримки. REST-запити вимагають постійного встановлення з'єднання та обробки кожного запиту окремо, що знижує ефективність при масштабуванні. GraphQL частково вирішує цю проблему, дозволяючи отримувати лише необхідні поля з бази даних, але все одно залежить від стабільності мережі та якості API зовнішнього сервісу.

API-синхронізація реалізується як класична модель запит–відповідь, у якій POS виступає клієнтом і ініціює кожну операцію отримання або відправлення даних. Кожен HTTP-виклик обмежений тайм-аутами на стороні клієнта й сервера, тому у разі деградації мережі або тимчасової недоступності сервісу доставки запити можуть завершуватися помилками чи повертати неповні результати. Додатковий рівень складності створюють механізми авторизації: використання

API-ключів, токенів доступу, схем на кшталт OAuth передбачає керування строком дії токенів, їх оновленням та безпечним зберіганням у POS або інтеграційних адаптерах. Одночасно платформи доставки застосовують обмеження швидкості (rate limiting), які встановлюють верхні межі кількості запитів за одиницю часу. При неправильному проєктуванні інтеграції це призводить до перевищення лімітів, отримання помилок і вимушених пауз у синхронізації. Окремою проблемою є так звані «chatty APIs», коли для обробки одного замовлення POS змушена виконувати багато дрібних запитів до різних endpoint-ів. За зростання навантаження це створює надмірний трафік, збільшує латентність і робить систему чутливою до будь-яких коливань продуктивності зовнішнього API.

У контексті інтеграції POS із службами доставки такі технічні обмеження прямо впливають на коректність обробки замовлень. Залежність від доступності зовнішнього API означає, що тимчасові збої або сплески затримок можуть призвести до ситуацій, коли замовлення створене на платформі доставки, але не встигає вчасно потрапити до POS, або, навпаки, повторно відправляється після тайм-ауту клієнта. За відсутності ідемпотентності endpoint-ів повторні виклики здатні спричиняти дублювання замовлень, некоректні зміни статусів і подвійні списання. Це вимагає від інтеграційного шару чіткої обробки HTTP-кодів помилок, продуманої стратегії повторних спроб, використання унікальних ідентифікаторів операцій і зберігання проміжних станів. У міру зростання кількості торгових точок і обсягу замовлень базова API-модель починає виявляти свої обмеження: POS вимушена обслуговувати велику кількість синхронних викликів, а централізована координація стає дедалі складнішою. Це природно підводить до необхідності переходу від суто запитно-відповідних схем до подійно-орієнтованих моделей, використання шини даних або гібридних рішень, у яких API виступає лише одним із рівнів інтеграції, а не єдиним механізмом синхронізації.

1.3.2 Синхронізація, керована подіями (event-driven model)

Подієва модель зазвичай реалізується на основі парадигми publish/subscribe, коли відправник події не звертається безпосередньо до конкретного сервісу-отримувача, а публікує повідомлення в спільну шину або тему. Зацікавлені компоненти підписуються на відповідні типи подій і обробляють лише ті, що відповідають їхній ролі. Такий підхід мінімізує залежності між сервісами, спрощує їхню еволюцію та дає змогу підключати нові підсистеми без зміни існуючих продюсерів. Важливим аспектом обробки подій є ідемпотентність: той самий запис має бути безпечним до повторної обробки, щоб дубльована подія не призводила до множинного списання товару чи створення кількох замовлень. Для коректної роботи у розподіленому середовищі події забезпечуються часовими мітками та, за потреби, механізмами упорядкування в межах певного ключа (наприклад, ідентифікатора замовлення), що дозволяє інтерпретувати їх у правильній послідовності навіть за паралельної доставки. На рівні інфраструктури застосовуються різні моделі гарантій доставки: at-most-once припускає можливу втрату подій, але виключає дублювання; at-least-once гарантує, що подія буде доставлена, хоча й потенційно кілька разів; exactly-once намагається поєднати гарантії повноти та відсутності дублювання, однак вимагає суттєво складніших протоколів і тісної взаємодії між брокером і споживачами.

У задачах POS–доставка подієвий підхід особливо ефективний через нерівномірний характер навантаження та різну швидкість оновлення різних типів даних. Статуси замовлень можуть змінюватися десятки разів на хвилину, тоді як оновлення меню або цін відбувається значно рідше, але має одразу бути відображене у всіх каналах. Подійна шина дозволяє обробляти велике число паралельних подій, не блокуючи основні транзакційні потоки в POS, а горизонтальне масштабування конс’юмерів та шардінг черг дають змогу підлаштовуватися під пікові періоди, не змінюючи бізнес-логіку. Водночас по мірі зростання кількості сервісів, тем подій і правил маршрутизації постає завдання централізованого управління схемами повідомлень, трансформаціями та політиками безпеки. Саме потреба впорядкувати подієвий обмін у великих гетерогенних IT-ландшафтах призвела до розвитку корпоративних шин даних

(ESB), які поєднують ідеї подійної взаємодії з більш жорстко структурованим підходом до маршрутизації, оркестрації й стандартизації повідомлень, що розглядається в наступному підрозділі.

1.3.3 Шина даних (ESB – Enterprise Service Bus)

Цей підхід набуває популярності серед компаній, які працюють у реальному часі. Тут кожна зміна в системі — нове замовлення, змінена ціна або оновлення статусу доставки — формує подію, яка надсилається в чергу повідомлень. POS-система, «підписана» на ці події, отримує їх і миттєво реагує. Така модель дає змогу уникнути постійних опитувань сервера, знижує навантаження на мережу та підвищує стабільність. Вона дозволяє працювати навіть при тимчасових збоях у зв’язку, оскільки всі події зберігаються у черзі (наприклад, у Apache Kafka, RabbitMQ або AWS SQS) і обробляються послідовно. Цей метод часто застосовується у великих мережах ресторанів і торговельних компаній, де важлива безперервна робота навіть за умов пікових навантажень.

Корпоративна шина даних виникла як відповідь на зростання кількості інформаційних систем у межах одного підприємства та потребу замінити хаотичні точкові інтеграції структурованою моделлю взаємодії. Замість численних парних зв’язків між POS, системами лояльності, ERP, складами, інтернет-магазинами та службами доставки впроваджується єдиний проміжний шар, через який проходить увесь обмін повідомленнями. Такий підхід дозволяє розглядати ESB як логічний «комунікаційний хаб», який відокремлює бізнес-додатки одне від одного і зменшує їхню взаємозалежність.

З технічного погляду ESB виконує декілька ключових функцій. По-перше, вона забезпечує трансформацію повідомлень: дані, що надходять у власному форматі конкретної системи, перетворюються до канонічної моделі, узгодженої для всього підприємства, а вже потім трансформуються до формату системи-отримувача. Це мінімізує кількість конвертаційних правил і спрощує супровід інтеграцій. По-друге, шина реалізує гнучку маршрутизацію, використовуючи різні патерни – за вмістом повідомлення, за типом операції, за

бізнес-правилами або часом доби. Такий рівень керованості дає змогу централізовано описувати, які події з POS повинні надходити до бек-офісу, які – до служби доставки, а які – в аналітичні системи.

Важливим елементом ESB є механізми валідації, безпеки та аудиту. Повідомлення проходять перевірку на структурну цілісність, відповідність обов’язковим полям, допустимість значень і права доступу. Єдина точка аутентифікації та авторизації для інтеграційних потоків дозволяє уніфікувати політики безпеки та відслідковувати всі операції обміну в централізованому журналі. Це особливо актуально для транзакцій, пов’язаних із оплатами, знижками або персональними даними клієнтів.

Порівняно з прямими API-інтеграціями, де кожна пара систем узгоджує власні протоколи, ESB забезпечує стандартизований спосіб взаємодії. Системи підключаються не одна до одної, а до шини, декларуючи, які типи повідомлень вони публікують і споживають. Це зменшує кількість інтеграційних зв’язків, спрощує додавання нових сервісів і дозволяє змінювати внутрішню реалізацію окремих систем без руйнування всієї інтеграційної схеми. Для POS це означає, що при підключенні нового сервісу доставки або нового бек-офісного модуля не потрібно модифікувати саму POS: достатньо налаштувати маршрути й трансформації на рівні ESB.

Окремий ефект від використання ESB полягає у перерозподілі навантаження. Шина бере на себе частину обчислень, пов’язаних із трансформацією, валідацією, логуванням і маршрутизацією, а також може реалізовувати буферизацію повідомлень, згладжуючи пікові навантаження на кінцеві системи. POS, у такій конфігурації, зосереджується на обробці бізнес-транзакцій, а не на керуванні великою кількістю зовнішніх підключень, що підвищує її стабільність та передбачуваність часу відгуку.

У контексті інтеграції POS із платформами доставки ESB дає змогу централізовано реалізувати правила мапування меню, модифікаторів, категорій, адрес і статусів замовлень. Наприклад, одне й те саме замовлення, сформоване клієнтом у зовнішньому сервісі, може одночасно передаватися до POS, системи

виробництва на кухні та аналітичного сховища, причому бізнес-логіка розподілу й трансформації реалізується на рівні шини. Це спрощує підтримку сценаріїв, коли заклад працює з декількома платформами доставки, але прагне зберігати єдині внутрішні довідники й моделі даних.

Водночас використання ESB має низку обмежень. У великих інфраструктурах, де через шину проходить більшість транзакцій, вона може перетворюватися на «вузьке місце», від продуктивності та доступності якого залежить робота всіх інтеграцій. Висока складність конфігурації, потреба у формалізації канонічної моделі даних і регламентів змін, а також концентрація інтеграційної логіки в одному компоненті роблять ESB відносно важковагомим рішенням порівняно з легкими подійно-орієнтованими інтеграціями або прямими API-зв’язками. Це вимагає зваженого підходу: у сценаріях із великою кількістю систем і вимогами до стандартизації ESB залишається виправданим інструментом, тоді як у більш гнучких, динамічних середовищах може поєднуватися з мікросервісними та подієвими підходами, які розвантажують шину та зменшують ризик формування єдиного критичного «ядра» інтеграційної архітектури.

1.3.4 Файлова або пакетна синхронізація (batch processing)

Файлова, або пакетна, синхронізація історично сформувалася як базовий підхід до обміну даними між ізольованими інформаційними системами, коли постійні мережеві з’єднання були дорогими або технічно недоступними. У старих POS-рішеннях вона стала стандартом завдяки простоті реалізації: касові термінали й бек-офісні системи працювали автономно впродовж дня, а передача накопичених транзакцій здійснювалася один чи кілька разів на добу у вигляді файлів. Це дозволяло розділити операційний контур і контур обробки даних, не навантажуючи каси онлайн-інтеграціями.

Технічно batch-підхід ґрунтується на періодичному формуванні експортних файлів у форматах CSV, XML, інколи власних бінарних форматах, їхньому перенесенні до цільової системи та подальшому імпорті. Розклади виконання визначаються за допомогою планувальників на кшталт cron, які запускають

сценарії формування, передачі й завантаження пакетів у визначені часові вікна. Для контролю цілісності широко застосовуються контрольні суми та службові записи в заголовках і «футерах» файлів, що дає змогу виявляти пошкодження при передачі або неповне формування набору даних. Буферизація на стороні джерела дозволяє накопичувати транзакції до моменту чергового вивантаження, зменшуючи кількість операцій обміну.

Перевагами такого підходу є відносна простота реалізації, низькі вимоги до інфраструктури та можливість роботи у відриві від постійного мережевого з’єднання. POS-система може повноцінно функціонувати офлайн, а передача даних до центрального вузла або сторонніх систем виконується у моменти доступності мережі. Саме тому batch-синхронізація й досі застосовується в невеликих локальних магазинах, закладах із нестабільним інтернет-з’єднанням, а також там, де сторонні системи не надають повноцінних API й підтримують лише імпорт/експорт файлів.

Недоліки пакетної синхронізації пов’язані з високою затримкою оновлень та низькою актуальністю даних у приймаючих системах. Помилки, допущені протягом періоду накопичення, виявляються лише під час обробки пакетів, що створює ризик їхнього масового тиражування. У мережевих закладах зростання кількості торгових точок і каналів продажу призводить до лавиноподібного збільшення обсягу файлів і складності їх координації, що ускладнює масштабування. У контексті POS та служб доставки пакетні оновлення можуть спричиняти ситуації, коли меню або ціни на платформі доставки залишаються застарілими до наступного циклу вивантаження, статуси замовлень відображаються з затримкою, а повторне формування файлів при збої призводить до дублювання вже оброблених замовлень.

У порівнянні із сучасними асинхронними подійно-орієнтованими підходами batch-процесинг розглядається як компромісне рішення, прийнятне для некритичних до часу задач — звітності, архівації, періодичного обміну агрегованими даними. Для оперативних сценаріїв, де важливі актуальність станів, синхронізація в реальному часі та коректна взаємодія з кількома сервісами

доставки, пакетна модель створює обмеження й виступає радше тимчасовим або допоміжним механізмом у загальній інтеграційній архітектурі.

1.3.5 Вебхуки та зворотні виклики (webhooks)

Ще один ефективний спосіб взаємодії — це вебхуки, які дозволяють зовнішній службі самостійно повідомляти POS про зміни. Наприклад, коли кур'єр приймає замовлення або клієнт його оплачує, служба доставки надсилає спеціальний HTTP-запит у POS-систему. Це дає змогу зменшити кількість постійних опитувань API і значно знизити навантаження на сервери.

Вебхуки часто використовуються разом із механізмами повторних спроб (retry), часових міток (timestamps) та контролю версій даних, щоб уникнути втрати або дублювання повідомлень у разі збою.

Для підвищення надійності взаємодії виклики вебхуків зазвичай доповнюються механізмами перевірки автентичності: у запиті передаються секретні ключі, токени або криптографічні підписи, які POS-система перевіряє перед обробкою події. Це дає змогу відсіяти неавторизовані або підроблені повідомлення та запобігти несанкціонованим змінам стану замовлень. Фактом успішної доставки події, як правило, вважається отримання коректної відповіді з кодуванням успіху з боку POS; за її відсутності служба доставки виконує повторні спроби надсилання. На відміну від pull-моделі, де ініціатором є POS, у випадку вебхуків діє push-схема: саме зовнішній сервіс вирішує, коли й що надсилати. Це створює додаткові вимоги до обробки одночасних та дубльованих викликів: endpoint-и POS мають коректно працювати за умов повторної доставки однієї й тієї самої події, не створюючи нових замовлень і не змінюючи стан об'єкта повторно. Вирішується це, зокрема, шляхом використання унікальних ідентифікаторів подій та ідемпотентної бізнес-логіки, яка гарантує, що повторне застосування тієї самої операції не змінює підсумкового результату.

У зв'язках між POS-системою та службами доставки такі особливості проявляються особливо виразно через нерівномірність навантаження: у пікові години кількість змін статусів стрімко зростає, і на POS може одночасно

надходити значний потік вебхуків. За умов мережевих затримок або тимчасових збоїв окремі події можуть приходити із запізненням або в іншому порядку, ніж вони фактично відбувалися, що створює ризики некоректної синхронізації. Якщо endpoint-и POS не є ідемпотентними, такі ситуації призводять до дублювання замовлень, неконсистентних статусів чи помилкових фінансових записів. Тому вебхуки, хоча й знижують загальне навантаження на API та роблять інтеграцію більш ефективною, висувають підвищені вимоги до спостережності системи: необхідні засоби відстеження успішності доставки подій, контролю затримок, виявлення розбіжностей у станах замовлень. Саме потреба системно бачити, які події були отримані, які втрачено, де виникли помилки трансформації або валідації даних, логічно підводить до завдання побудови повноцінного моніторингу й контролю обміну, що розглядається у наступному підрозділі.

1.3.6 Моніторинг і контроль обміну даними

Моніторинг і контроль обміну даними є критичними компонентами будь-якої інтеграційної архітектури, у якій взаємодіють кілька автономних систем. Для зв’язки POS–адаптери–API служб доставки–брокер подій саме моніторинг забезпечує можливість своєчасно виявляти збої, затримки та десинхронізацію, які не є очевидними на рівні окремих компонентів. Синхронізація в розподіленому середовищі завжди супроводжується ризиками втрати чи дублювання подій, накопичення черг і деградації продуктивності, тому без постійного спостереження за станом інтеграцій неможливо гарантувати цілісність даних та стабільність бізнес-процесів.

У контексті інтеграції POS із платформами доставки доцільно розрізняти технічний і прикладний моніторинг. Технічний моніторинг охоплює контроль доступності сервісів, стану черг у брокері, часу відповіді зовнішніх API, обсягу та швидкості обробки повідомлень. Він дає змогу виявляти перевантаження, зростання латентності, деградацію продуктивності адаптерів або нестабільність мережевих з’єднань. Прикладний моніторинг зосереджений на змісті й коректності самих даних: відповідності статусів замовлень між POS і платформою

доставки, узгодженості сум, коректності трансформації меню та цін, відсутності систематичних помилок у бізнес-логіці.

Механізми контролю обміну даними базуються на поєднанні журналювання, трасування подій, збору метрик і систем попереджувальних сповіщень. Журнали фіксують ключові дії інтеграційних компонентів, зміст отриманих і переданих повідомлень, коди помилок і виняткові ситуації, що дозволяє відтворити ланцюжок подій при розслідуванні інцидентів. Трасування дає змогу простежити шлях конкретного замовлення або події через усі задіяні сервіси, пов’язуючи записи логів за кореляційними ідентифікаторами. Метрики продуктивності – час обробки замовлень, довжина черг, частота помилок, частка повторних спроб – формують кількісну основу для дашбордів та аналітики. Системи alerting генерують сповіщення при виході показників за допустимі межі, що дає можливість реагувати до того, як проблема стане помітною клієнтам.

Такі інструменти дозволяють виявляти широкий спектр аномалій у роботі інтеграцій. Раптове зростання затримок у передачі статусів може свідчити про перевантаження брокера або деградацію зовнішнього API. Збільшення кількості дубльованих замовлень вказує на помилки у механізмах повторної доставки або недостатню ідемпотентність операцій. Розбіжності в кількості чи станах замовлень між POS і платформою доставки, що фіксуються прикладним моніторингом, сигналізують про збої під час трансформації даних, некоректну фільтрацію або часткову втрату подій. Без системного моніторингу такі проблеми можуть тривалий час залишатися непоміченими, накопичуючись у вигляді фінансових розбіжностей і конфліктних ситуацій з клієнтами.

Важливим аспектом є формалізація та контроль показників SLA і SLO у взаємодії з сервісами доставки. Рівень доступності зовнішніх API, середній час відповіді, максимальні допустимі затримки при створенні замовлення або оновленні статусу мають бути формалізовані та технічно відслідковувані. Це дозволяє відрізняти внутрішні проблеми інтеграційної платформи від збоїв на стороні партнера, обґрунтовувати вимоги до якості сервісу та приймати рішення щодо маршрутизації потоку замовлень між різними агрегаторами.

Централізований моніторинг, реалізований у вигляді інтеграційних дашбордів, дає змогу отримувати цілісне уявлення про стан усієї зв’язки POS–адаптери–брокер–служби доставки в реальному часі. На одному екрані можуть відображатися ключові технічні й прикладні показники, історія інцидентів і поточні сповіщення, що скорочує час виявлення та локалізації проблеми. Операційні команди отримують можливість проактивно реагувати на відхилення – тимчасово обмежувати навантаження, переключати потоки, коригувати розклади, запускати процедури повторної синхронізації.

Відсутність системного моніторингу призводить до неконтрольованої десинхронізації даних, коли розбіжності між POS і службами доставки стають помітними лише через скарги клієнтів, невідповідності у фінансових звітах або результати вибіркових перевірок. У такій ситуації кожен інцидент вимагає ручного розслідування, а причини помилок часто залишаються неусунутими. Це підриває довіру до інтеграційного рішення, збільшує операційні витрати та обмежує можливості масштабування. Саме тому моніторинг і контроль обміну даними мають розглядатися як невід’ємна частина архітектури інтеграцій, а не як додатковий сервіс, що впроваджується «за залишковим принципом».

1.4 Сучасні технології інтеграції POS та зовнішніх систем доставки

Еволюція POS-систем значною мірою була зумовлена впровадженням інтеграційних технологій. Раніше основний обмін даними відбувався за допомогою локальних баз даних або обмежених API, то сьогоднішні архітектури наголошують на гнучкості, масштабованості та безперервному обміні інформацією між усіма цифровими компонентами бізнесу.

Одним із ключових напрямків розвитку є застосування мікросервісної архітектури. У цій моделі кожен модуль системи — обробка замовлень, управління меню, облік запасів, фінансова звітність — функціонує як окремий сервіс із власним API. Це означає, що будь-який компонент можна оновлювати або масштабувати незалежно від інших, без зупинки всієї системи. Такий підхід

особливо цінний для підприємств із великою кількістю філій або високим навантаженням у пікові години.

Для забезпечення стабільної синхронізації між цими сервісами застосовують брокери повідомлень, такі як Apache Kafka, RabbitMQ або MQTT. Вони дозволяють обробляти тисячі подій у реальному часі — наприклад, нові замовлення, зміни цін або оновлення статусу доставки — без втрати даних навіть при тимчасових збоях у мережі.

Інша тенденція — використання API-шлюзу — проміжного шару, який обробляє всі запити від клієнтів до внутрішніх сервісів. API-шлюз забезпечує централізовану авторизацію, контроль доступу, кешування, моніторинг трафіку та трансформацію формату запитів. Це знижує навантаження на важливі модулі основної системи та підвищує безпеку.

У великих мережах ресторанів або торгових компаніях API-шлюз дозволяє гнучко керувати потоками даних між десятками філій і зовнішніх сервісів. Наприклад, у разі перевантаження одного регіонального вузла, шлюз автоматично перенаправляє трафік до резервного сервера, що гарантує безперебійну роботу POS.

Платформи проміжного програмного забезпечення (Omnivore, Blend, RestoAPI тощо) активно інтегруються в ресторанний бізнес, будучи загальним посередником між POS-системою та деякими зовнішніми сервісами. Вони дозволяють виконувати інтеграцію за загальними схемами даних без повної зміни коду рішення POS. JSON та XML є двома з найбільш широко використовуваних інструментів передачі даних, оскільки в більшості мов програмування їх можна легко обробляти. Протоколи Buffers (gRPC) застосовуються для швидших, більш легких API-з'єднань, зберігаючи цілісність структури даних.

В останні роки спостерігається активне впровадження технологій контейнеризації — зокрема, Docker та Kubernetes. Вони дозволяють ізолювати кожен мікросервіс, швидко розгортати оновлення і масштабувати ресурси системи відповідно до поточного навантаження. Це робить архітектуру POS-систем ще більш гнучкою та надійною.

Однією з найважливіших функцій у сучасних системах є захист обміну даними. Через шифрування HTTPS, токени автентифікації (OAuth 2.0, JWT) та контроль ролей користувачів відбувається безпечна комунікація в різних хмарних розгортаннях та середовищах.

Особливу увагу сьогодні приділяють дотриманню міжнародних стандартів безпеки — таких як PCI DSS для платіжних систем або GDPR для обробки персональних даних. Це дозволяє компаніям працювати з глобальними сервісами доставки, не порушуючи вимог щодо зберігання та обробки клієнтської інформації.

Також потрібно згадати про зростання хмарних інтеграційних сервісів у формі iPaaS (Integration Platform as a Service), включаючи MuleSoft CloudHub, Zapier, Boomi. Вони підтримують конфігурацію взаємодії між системами без прямого програмування та використання графічних інтерфейсів для встановлення потоку даних. Це ті рішення, які дозволяють організаціям швидко реагувати на зміни ринкових умов без витрат на дорогий та тривалий розвиток.

Завдяки iPaaS навіть невеликі компанії можуть впроваджувати професійні інтеграційні рішення без власної IT-команди, використовуючи візуальні інструменти для побудови процесів і готові конектори для популярних служб доставки.

Таким чином, сучасні інтеграційні технології зосереджені не лише на швидкості, але й на стабільності, контролі помилок та можливості масштабування без зупинки сервісу. Це закладає основу для побудови надійної взаємодії для системи POS та служб доставки.

1.5. Проблеми ефективності взаємодії та напрями удосконалення

Попри стрімкий розвиток технологій, інтеграція POS-систем і зовнішніх сервісів доставки залишається складним і багатогранним процесом, який потребує постійного вдосконалення. Більшість проблем пов’язані не лише з технічними

обмеженнями, але й з відсутністю єдиних стандартів обміну даними, високими вимогами до безпеки та стабільності роботи систем у реальному часі.

Нестандартність форматів даних. Кожна платформа доставки має власну структуру API та набір методів взаємодії. Через це розробникам POS-систем доводиться створювати окремі інтеграційні модулі для кожного партнера, що суттєво збільшує витрати на підтримку, оновлення та тестування.

Рішенням може бути впровадження уніфікованих схем обміну даними або використання адаптерів, які автоматично перетворюють дані у потрібний формат. У світовій практиці вже існують спроби стандартизації, наприклад, створення відкритих API-специфікацій для ресторанної індустрії. У перспективі саме відкриті стандарти дозволять скоротити час інтеграції з новими партнерами з кількох тижнів до кількох годин.

Затримки в обміні інформацією. Навіть при використанні REST- або GraphQL-API частина запитів обробляється із затримками через перевантаження мережі, недосконалу архітектуру або різницю у часових зонах серверів. Це призводить до лагів у прийомі замовлень, помилкових статусів або дублювання інформації.

Для зменшення таких затримок застосовують асинхронну передачу даних, кешування часто використовуваних запитів і механізми повторних спроб (retry) з контрольованими таймаутами. Також ефективним підходом є впровадження CDN (Content Delivery Network) — це дає змогу розподіляти навантаження між кількома географічними вузлами та забезпечувати більш стабільну швидкість обробки запитів.

Відсутність повної консистентності даних. У розподілених системах замовлення часто проходить через кілька незалежних сервісів — POS, CRM, службу доставки, систему лояльності. Будь-яка затримка або збій на одному етапі може призвести до розбіжностей у статусах.

Для таких випадків застосовуються архітектурні патерни event sourcing та CQRS (Command Query Responsibility Segregation). Вони дозволяють не лише зберігати всі зміни у вигляді подій, а й відновлювати стан системи у разі збою.

Крім того, для контролю консистентності даних дедалі частіше використовуються блокчейн-технології, які дають змогу фіксувати історію транзакцій у незмінному вигляді.

Обмежена відмовостійкість. Інтеграційна інфраструктура POS і сервісів доставки повинна залишатися працездатною навіть за часткових збоїв. У разі відключення одного вузла або сервісу система не має повністю зупинятись. Для цього застосовують кластеризацію, реплікацію баз даних і паралельну обробку подій.

Крім того, важливим елементом є механізм circuit breaker — автоматичне відключення несправного компонента для запобігання «ефекту доміно», коли збій одного сервісу викликає падіння всієї системи. Додатково сучасні архітектури включають балансувальники навантаження (Load Balancers) і автоматичні механізми масштабування, які підвищують стійкість системи при пікових навантаженнях.

Безпека інтеграційних з’єднань. Безпека — одна з найкритичніших проблем у сучасній екосистемі інтеграцій. Передача даних через відкриті мережі створює ризики перехоплення, підробки запитів або несанкціонованого доступу до конфіденційної інформації клієнтів. Необхідно впроваджувати шифрування каналів зв’язку (HTTPS, SSL/TLS), автентифікацію через токени (OAuth 2.0, JWT), а також контроль ролей користувачів. Концепція «zero trust» передбачає, що кожен запит перевіряється незалежно від його джерела. У великих мережах ресторанного бізнесу це доповнюється системами моніторингу безпеки (SIEM) та автоматичним виявленням аномальної активності (IDS/IPS).

Людський фактор і складність підтримки. Інтеграція систем часто залежить від кваліфікації технічного персоналу. Будь-які зміни на стороні зовнішнього сервісу можуть викликати збої у роботі POS або втрату зв’язку.

Тому важливо впроваджувати системи автоматичного моніторингу, сповіщень (alerting) та централізоване журналювання (logging). Сучасні DevOps-практики, такі як CI/CD (Continuous Integration / Continuous Deployment), дозволяють швидко

тестувати й розгортати оновлення без втручання людини, що значно зменшує ризик помилок.

Напрями подальшого вдосконалення. Подальший розвиток інтеграційних рішень передбачає створення єдиного міжнародного стандарту взаємодії між POS-системами та платформами доставки. Це може бути реалізовано у вигляді відкритого протоколу або SDK, який спрощуватиме підключення нових партнерів і зменшить витрати на підтримку сумісності.

Додатково перспективним напрямом є застосування штучного інтелекту та машинного навчання. Такі системи здатні аналізувати мережеві затримки, прогнозувати навантаження, виявляти ризики збоїв і оптимізувати маршрути передачі даних. Наприклад, алгоритми можуть автоматично визначати найменш завантажений сервер доставки або прогнозувати пікові години для ефективного розподілу запитів.

Ще одним важливим напрямом розвитку є перехід до концепції «інтелектуальної інтеграції» (Intelligent Integration) — коли системи не лише обмінюються інформацією, а й здатні самостійно приймати рішення: від адаптації під зміни API-партнера до автоматичного відновлення зв’язку після збою.

2 МОДЕЛЮВАННЯ ТА ОБҐРУНТУВАННЯ МЕТОДУ ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ ІНТЕГРАЦІЇ POS-СИСТЕМ І СЛУЖБ ДОСТАВКИ

2.1. Постановка задачі та вимоги до системи синхронізації

Аналіз сучасних POS-систем, технологій інтеграції та моделей обміну даними, наведений у розділі 1, показав, що зростання обсягів онлайнових замовлень і поширення служб доставки формують підвищені вимоги до швидкості, надійності та узгодженості даних між внутрішніми обліковими системами закладу та зовнішніми платформами. Наявні підходи до інтеграції часто реалізуються у вигляді точкових рішень, жорстко прив’язаних до конкретних API, не забезпечують гарантованої доставки подій, мають обмежені можливості масштабування та ускладнюють супровід при підключенні нових сервісів.

У цих умовах постає задача розроблення методу синхронізації даних між POS-системами та службами доставки, який забезпечує своєчасну та цілісну передачу інформації про меню, замовлення та їх статуси, зменшує кількість помилок ручного введення, скорочує затримки обробки замовлень і полегшує розширення інтеграційної інфраструктури. Об’єктом дослідження є процес інтеграції POS-систем із зовнішніми службами доставки, а предметом – метод синхронізації даних на основі подієво-орієнтованої моделі та інтеграційного шару.

Для формалізації задачі введемо такі базові вимоги до проектованої системи синхронізації.

Функціональні вимоги:

1. Система має забезпечувати двосторонній обмін даними між POS-системою та множиною служб доставки, підтримуючи паралельну роботу з кількома зовнішніми платформами.
2. Має бути реалізовано механізм синхронізації меню: передавання номенклатури, цін, доступності позицій, а також фіксація змін (створення, оновлення, деактивація товарів) у єдиному форматі даних.

3. Система повинна приймати замовлення із зовнішніх служб доставки, трансформувати їх у внутрішній формат POS-системи та маршрутизувати до відповідних модулів обліку і виробництва.
4. Необхідно забезпечити синхронізацію життєвого циклу замовлення (створення, підтвердження, приготування, видача, доставка, скасування) між POS та службами доставки з фіксацією всіх проміжних статусів.
5. Система має підтримувати обробку змін і скасувань замовлень, включно з частковими змінами складу, адреси чи часу доставки.
6. Повинні бути реалізовані засоби журналювання та моніторингу обміну даними, що дають змогу відслідковувати історію операцій, виявляти збої й аналізувати причини помилок.

Нефункціональні вимоги:

1. Час доставки критичних подій (створення та зміна статусу замовлення) має задовольняти задані обмеження затримки, що визначаються бізнес-вимогами (рівень сервісу для клієнта та операційні регламенти закладу).
2. Система повинна забезпечувати надійність та відмовостійкість: коректну обробку помилок мережі й зовнішніх сервісів, механізми повторних спроб (retry) та запобігання дублюванню подій.
3. Архітектура синхронізації має бути масштабованою, з можливістю збільшення пропускної здатності при зростанні кількості замовлень та підключенні нових служб доставки без суттєвих змін бізнес-логіки.
4. Інтеграційний шар повинен бути розширюваним, тобто підтримувати додавання нових адаптерів до сторонніх API за рахунок конфігурації та повторного використання спільних компонентів.
5. Має бути забезпечено базовий рівень інформаційної безпеки: автентифікація та авторизація взаємодіючих систем, захищена передача даних та контроль цілісності повідомлень.

Таким чином, постановка задачі полягає у побудові такої моделі та програмної реалізації системи синхронізації, яка, з одного боку, задовольняє наведені функціональні й нефункціональні вимоги, а з іншого – дозволяє кількісно оцінити

та покращити показники ефективності інтеграції «POS – служби доставки» порівняно з наявними підходами, розглянутими у розділі 1.

2.2. Концептуальна модель інтегрованої системи «POS – служба доставки»

На основі сформульованих у підрозділі 2.1 вимог побудуємо концептуальну модель інтегрованої системи «POS – служба доставки», яка відображає основні підсистеми, їхні ролі та потоки даних між ними. Така модель слугує логічною основою для подальшого формального моделювання, вибору архітектурних рішень та проектування алгоритмів синхронізації. У загальному вигляді структуру системи можна подати у вигляді трьох рівнів: внутрішня система автоматизації закладу (POS), інтеграційний шар та зовнішні служби доставки. На рисунку 2.1 подано концептуальну модель інтегрованої системи «POS – служба доставки», що відображає ключові компоненти, їхні функції та напрямки потоків даних.

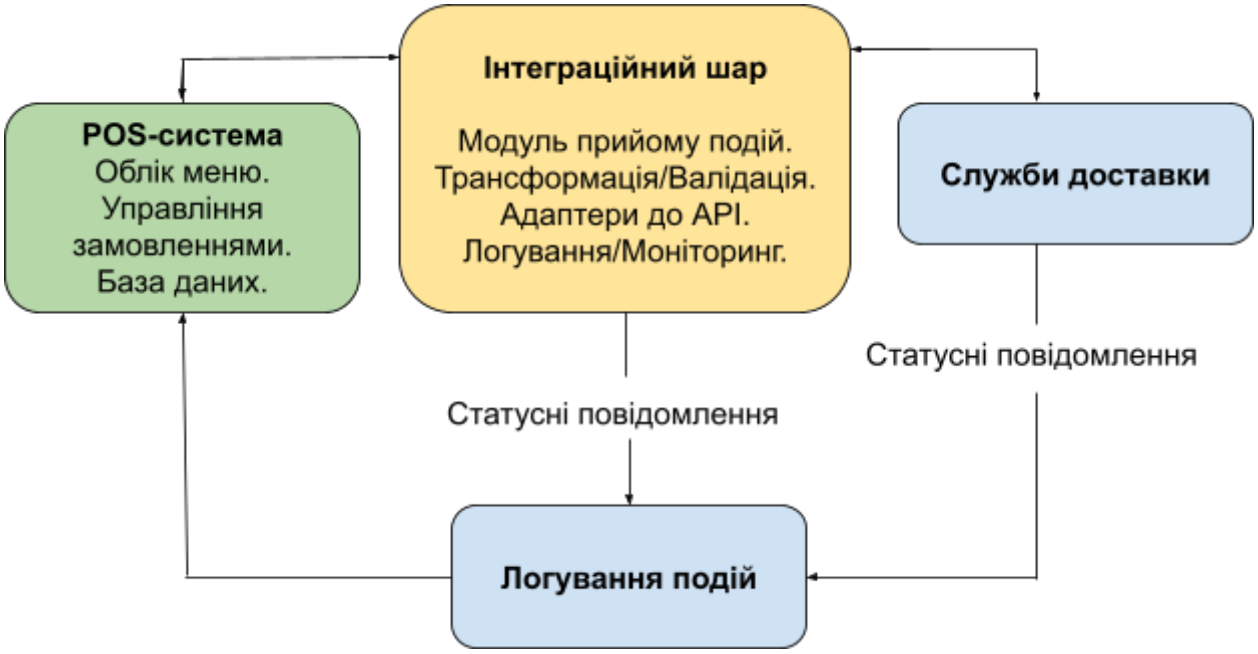


Рис. 2.1 Концептуальна модель інтегрованої системи “POS - служба доставки”

До складу POS-системи входять модулі обліку номенклатури (меню), управління замовленнями, обліку оплат та взаємодії з касовим обладнанням. У контексті інтеграції її можна розглядати як джерело «еталонних» даних про меню

та як внутрішній реєстр замовлень, у якому фіксуються всі операції з моменту створення замовлення до його закриття. POS-система зберігає інформацію в локальній базі даних та надає інтерфейси (API або внутрішні сервіси) для читання й запису даних, якими користується інтеграційний шар.

Інтеграційний шар виступає проміжною підсистемою між POS та зовнішніми службами доставки. Його ключовими функціями є: нормалізація даних до єдиного проміжного формату, маршрутизація запитів і подій, адаптація протоколів та форматів до вимог конкретних служб доставки, а також забезпечення надійності та відмовостійкості обміну. Логічно інтеграційний шар включає модулі прийому подій з POS, модуль трансформації та валідації повідомлень, адаптери до API служб доставки, модуль обробки відповідей і статусів, а також підсистему журналювання й моніторингу. Саме на рівні інтеграційного шару реалізується подієво-орієнтована модель синхронізації, обрана в роботі як базова.

Служби доставки у концептуальній моделі представлені як множина зовнішніх систем із власними API, моделями даних і бізнес-логікою. Кожна служба виступає споживачем інформації про меню та статуси замовлень, а також джерелом подій створення, оновлення чи скасування замовлень. У загальному випадку вони взаємодіють з інтеграційним шаром через HTTP/REST API, webhooks або інші визначені протоколи, при цьому специфіка кожної платформи приховується за відповідним адаптером.

У межах концептуальної моделі можна виділити кілька основних потоків даних. Перший – це потік синхронізації меню: зміни у довіднику товарів, цін, доступності та акцій, що вносяться в POS-системі, передаються до інтеграційного шару, де перетворюються у проміжний формат і розповсюджуються до підключених служб доставки. Другий потік – потік вхідних замовлень: замовлення, створені клієнтами у мобільних застосунках або веб-інтерфейсах служб доставки, надходять до інтеграційного шару, нормалізуються та реєструються у POS як внутрішні замовлення із збереженням усіх необхідних атрибутів (склад, суми, адреса, часові вікна доставки тощо).

Третій ключовий потік – потік статусних повідомлень. Зміни стану замовлення, що відбуваються у POS (підтвердження, взяття в роботу, готовність,

видача кур’єру), мають бути відображені у відповідній службі доставки, а зміни, ініційовані з боку служби доставки (призначення кур’єра, початок доставки, доставка, скасування), – відображені у POS. Інтеграційний шар забезпечує узгодженість життєвого циклу замовлення, перетворюючи внутрішні події POS у зовнішні статуси та навпаки.

Окремо виділяється потік логування та моніторингу, що включає записи про всі запити, відповіді, помилки та повторні спроби обробки подій. Дані цього потоку накопичуються в окремому сховищі журналів та метрик і використовуються для виявлення збоїв, аналізу продуктивності, а також побудови звітів щодо якості синхронізації. Наявність такого потоку є необхідною умовою для подальшої кількісної оцінки ефективності інтеграції, яка проводиться у розділі 3.

Таким чином, концептуальна модель інтегрованої системи «POS – служба доставки» визначає склад основних підсистем, ролі інтеграційного шару та структуру критичних потоків даних. На її основі в наступних підрозділах буде сформовано формалізовану модель процесів синхронізації, а також розроблено алгоритми обробки подій, що реалізують запропонований метод підвищення ефективності інтеграції.

2.3 Формалізована модель процесів синхронізації

Для подальшого аналізу ефективності інтеграції розглянемо процеси синхронізації між POS-системою та службами доставки як послідовність обробки подій у кількох вузлах системи. До таких подій належать: створення нового замовлення, зміна його статусу (підтверджено, в роботі, готово, доставлено, скасовано), а також оновлення меню (зміна цін, наявності позицій тощо). Кожна подія послідовно проходить через кілька етапів: реєстрацію в POS, обробку в інтеграційному шарі, передачу до зовнішньої служби доставки та отримання зворотної відповіді. На рисунку 2.2 зображено формалізовану модель синхронізації, яка враховує час обробки, затримки, ризики втрат і дублювань подій між основними вузлами системи.

Рис. 2.2 Формалізована модель процесів синхронізації POS-систем та служб доставок

Для опису роботи системи виділимо ключові параметри. По-перше, це інтенсивність надходження подій – скільки замовлень та змін статусів надходить за одиницю часу. У години пік інтенсивність різко зростає, що створює додаткове навантаження на всі компоненти.

По-друге, важливим параметром є час обробки події в кожному вузлі: скільки часу POS витрачає на фіксацію замовлення, скільки – інтеграційний шар на трансформацію й маршрутизацію, і скільки часу займає відповідь зовнішнього API.

До цього додається мережна затримка – час, який подія «проводить у дорозі» між компонентами. Нарешті, мають значення ймовірність втрати події (коли замовлення або зміна статусу не доходить до пункту призначення) та ймовірність дублювання (коли один і той самий запит обробляється двічі, наприклад через повторні відправлення при збоях).

На основі цих параметрів формуються цільові показники якості системи синхронізації, за якими оцінюється ефективність запропонованого методу. До таких показників належать:

1. Середній час обробки замовлення – від моменту його створення до відображення у POS та/або службі доставки в актуальному статусі;
2. Середній час оновлення статусу – наскільки швидко зміни стану замовлення стають видимими у всіх залучених системах;
3. Частка втрачених подій – відсоток замовлень або статусів, які не були доставлені до кінцевої системи;
4. Частка дубльованих подій – скільки разів одне й те саме замовлення або зміна статусу з’являється в системі повторно;
5. Навантаження на окремі вузли – ступінь завантаження POS, інтеграційного шару, брокера повідомлень та адаптерів до служб доставки;

6. Доступність інтеграційного шару – частка часу, протягом якого інтеграція працює коректно та приймає події.

Взаємозв’язок між параметрами та цими показниками є прямим і зрозумілим. Якщо інтенсивність надходження замовлень зростає, а ресурси системи залишаються незмінними, виникають черги на обробку, збільшується середній час обробки та зростає ризик втрати подій. Якщо час обробки в окремих компонентах занадто великий або з’являються значні мережні затримки, користувачі бачать «застарілі» статуси замовлень, що безпосередньо впливає на якість сервісу. Висока ймовірність збоїв на стороні зовнішніх API без належних механізмів повторних спроб призводить до втрати замовлень, а нераціональна реалізація цих механізмів – до дублювання.

Запропонований у роботі подієво-орієнтований метод синхронізації з використанням інтеграційного шару та брокера повідомлень спрямований на покращення саме цих показників. Асинхронна обробка подій та черги повідомлень дозволяють зменшити час очікування при пікових навантаженнях і рівномірніше розподіляти навантаження між вузлами. Централізоване журналювання та контроль ідентифікаторів подій знижують ймовірність дублювання, а механізми гарантованої доставки та повторних спроб – ймовірність втрати подій при тимчасових збоях зовнішніх сервісів. У подальших підрозділах та в розділі 3 на основі цих показників буде проведено кількісний аналіз і порівняння роботи системи до та після впровадження запропонованого методу синхронізації.

2.4 Аналіз критичних параметрів та «вузьких місць» системи

На основі формалізованої моделі процесів синхронізації можна виділити низку параметрів, які мають найбільший вплив на цільові показники якості системи – час обробки замовлень, узгодженість статусів, рівень втрат та дублювання подій. До таких критичних параметрів належать: довжина та час очікування в чергах обробки подій, частота обривів з’єднань з зовнішніми службами доставки, стабільність роботи брокера повідомлень, швидкодія адаптерів до API та продуктивність бази даних, що використовується інтеграційним шаром. Саме вони формують «вузькі місця», де події накопичуються, втрачаються або обробляються повторно.

Одним із ключових вузьких місць є черги на обробку подій в інтеграційному шарі. За підвищених навантажень (години пік, акційні періоди) збільшення інтенсивності надходження замовлень призводить до накопичення подій у чергах, зростання часу очікування та, як наслідок, до затримки оновлення статусів у POS і на платформах доставки. Якщо черги не мають належних обмежень та механізмів керування пріоритетами, це може спричиняти ситуації, коли не критичні події (наприклад, оновлення меню) конкурують за ресурси з критичними (створення та зміна статусів замовлень).

Другим суттєвим вузьким місцем є нестабільність мережевої взаємодії з зовнішніми службами доставки: обриви з’єднань, перевищення тайм-аутів, тимчасова недоступність API. У традиційних реалізаціях такі збої часто призводять до втрати запитів або до ручного повторного введення замовлень. Якщо механізми повторних спроб реалізовані без контролю ідентичності подій, це створює ризик дублювання замовлень, що безпосередньо погіршує якість сервісу для кінцевого користувача та ускладнює облік.

Окрему групу критичних параметрів формують характеристики брокера повідомлень та сховища даних. Недостатня пропускна здатність брокера або неправильно налаштовані параметри обробки черг (розмір партій, кількість споживачів, політика повторних спроб) можуть спричиняти додаткові затримки, а у гіршому випадку – втрату повідомлень при відмовах. Аналогічно, повільні

операції запису в базу даних інтеграційного шару збільшують загальний час обробки подій та можуть призводити до тимчасової недоступності сервісів, що працюють поверх цієї бази.

Таким чином, для підвищення ефективності інтеграції необхідно сфокусуватися на покращенні трьох груп характеристик системи:

– затримки обробки подій – шляхом оптимізації черг, розділення потоків на критичні та некритичні, асинхронної обробки та масштабування вузлів;

– надійності доставки подій – за рахунок впровадження гарантованої доставки, контрольованих повторних спроб, ідемпотентності операцій та унікальних ідентифікаторів подій;

– відмовостійкості інтеграційного шару – через застосування резервування, механізмів автоматичного відновлення, моніторингу стану компонентів і завчасного виявлення деградації продуктивності.

Запропонований у роботі метод синхронізації орієнтується саме на ці критичні параметри: він покликаний зменшити середню та пікову затримку обробки замовлень, мінімізувати втрати та дублювання подій при збоях зовнішніх сервісів і забезпечити стабільну роботу інтеграційного шару за умов зростання навантаження. У наступних підрозділах описуються архітектурні рішення та алгоритми, які реалізують зазначені покращення.

Додатково до виділених критичних параметрів доцільно розглянути їхній вплив не лише на технічні, а й на бізнес-показники системи. Затримки в обробці подій безпосередньо трансформуються в затримки у підтвердженні замовлень для клієнтів, що зменшує ймовірність їх завершення й підвищує частку скасувань. Нестабільність синхронізації статусів призводить до ситуацій, коли клієнт у мобільному застосунку бачить «заморожений» стан замовлення, а персонал закладу працює за іншою картиною. Це збільшує навантаження на кол-центри, спричиняє конфлікти між службою доставки та рестораном і в результаті негативно впливає на репутацію бренду. Тому кожен технічний параметр – довжина черги, коефіцієнт завантаження вузла, частота відмов API – має розглядатися у зв’язку з операційними метриками: середнім часом обслуговування замовлення, відсотком скасувань, кількістю звернень від клієнтів.

Окремий клас «вузьких місць» пов’язаний із невідповідністю моделей даних POS-системи та служб доставки. Навіть за формально коректної роботи API можуть виникати логічні неконсистентності: різна інтерпретація статусів, відмінні правила округлення сум, різна обробка модифікаторів страв. У таких випадках джерелом проблем стає не затримка чи відмова, а саме трансформація даних на інтеграційному рівні. Якщо цей процес не формалізований і реалізований у вигляді розрізнених «латок», інтеграційний шар фактично перетворюється на ще одне «вузьке місце», де накопичуються помилки, які важко діагностувати. Виявлення таких логіко-семантичних вузьких місць вимагає чіткої моделі мапінгу бізнес-сутностей і однозначних правил трансляції статусів, що також впливає на цілісність процесів синхронізації.

Не менш критичною є політика управління ресурсами на рівні інтеграційного шару. За відсутності обмежень на розмір черг, механізмів відбраковування надмірних або некоректних подій і чітких тайм-аутів інтеграційна система ризикує деградувати «тихо»: накопичення великої кількості застарілих подій призводить до зростання часу обробки, зростає навантаження на базу даних та брокер повідомлень, ускладнюється аналіз логів. Це створює ілюзію працездатності за низького навантаження і різку деградацію у годину пік. Тому до критичних параметрів слід віднести також політики утилізації застарілих подій, обмеження на розмір та час життя повідомлень у чергах, а також наявність механізмів пріоритезації, коли критичні бізнес-події (створення замовлення, зміна фінального статусу) обробляються раніше за некритичні (фонове оновлення довідників).

Сукупний аналіз технічних і бізнес-параметрів дозволяє сформувати узагальнений профіль ризиків для інтеграційної системи. До нього належать ризики втрати замовлень при пікових навантаженнях, накопичення невідправлених подій при відмовах зовнішніх API, розсинхронізація статусів, перевантаження окремих компонентів та логіко-семантичні помилки при трансформації даних. Саме ці аспекти мають бути враховані у подальшому проектуванні методу синхронізації. У наступному підрозділі обґрунтовується вибір подієво-орієнтованої моделі як способу системного впливу на зазначені критичні параметри: через розвантаження вузлів за рахунок асинхронної обробки,

використання стійких черг із керованими політиками повторних спроб, формалізацію мапінгу даних та централізоване журналювання всіх подій, що проходять через інтеграційний шар.

2.5 Обґрунтування вибору подієво-орієнтованого методу синхронізації

У розділі 1 було розглянуто кілька поширених підходів до інтеграції інформаційних систем: пакетна (batch) синхронізація, пряма API-орієнтована взаємодія, інтеграція на основі шини даних (ESB) та використання вебхуків (webhooks). Пакетна синхронізація є прийнятною для періодичного оновлення довідників або аналітичних даних, однак вона не забезпечує потрібної оперативності для обробки онлайн-замовлень та змін статусів у реальному часі. Пряма API-орієнтована інтеграція, коли кожна подія обробляється одноразовим запитом до зовнішнього сервісу, призводить до жорсткого зв’язування між системами, ускладнює підключення нових служб доставки та погано масштабується за умов зростання навантаження.

Використання ESB та класичних інтеграційних шин дозволяє централізувати маршрутизацію повідомлень, але такі рішення часто є важкими у впровадженні, потребують значних ресурсів і не завжди оптимізовані для високочастотних транзакцій типу «замовлення–статус». Механізм вебхуків, який широко застосовується службами доставки, забезпечує подієві сповіщення, але у базовому вигляді не вирішує задачі гарантованої доставки, ідемпотентності та централізованого контролю черг подій. Кожен із зазначених підходів частково відповідає окремим вимогам, визначеним у підрозділі 2.1, проте не забезпечує комплексного виконання вимог щодо швидкодії, масштабованості, відмовостійкості та розширюваності інтеграційного шару.

Подієво-орієнтована модель синхронізації, обрана як основа запропонованого методу, передбачає представлення всіх змін стану системи у вигляді потоку подій, що реєструються, зберігаються та обробляються асинхронно через спеціалізований брокер повідомлень. Такий підхід краще відповідає вимогам з підрозділу 2.1, оскільки дозволяє розділити прийом подій від POS та служб доставки і їх подальшу обробку, тим самим зменшуючи затримку реакції на

нові замовлення та зміни статусів. Крім того, використання централізованого потоку подій дає можливість гнучко масштабувати окремі споживачі подій без зміни бізнес-логіки, що критично важливо за умов зростання інтенсивності замовлень.

З точки зору критичних параметрів, визначених у підрозділі 2.4, подієво-орієнтована модель дозволяє зменшити час очікування в чергах за рахунок паралельної обробки подій, мінімізувати втрати завдяки використанню стійких черг та механізмів повторних спроб, а також контролювати дублювання за допомогою ідентифікаторів подій та ідемпотентності операцій на стороні споживачів. Окремий брокер повідомлень з функціями журналювання забезпечує додатковий рівень відмовостійкості: у разі тимчасової недоступності однієї з підсистем події накопичуються у чергах і обробляються після відновлення сервісу, не втрачаючись і не потребуючи ручного втручання.

Основні принципи запропонованого методу синхронізації полягають у наступному: представлення всіх бізнес-подій (створення та зміна статусу замовлень, оновлення меню) у вигляді уніфікованих повідомлень; використання інтеграційного шару як єдиної точки входу та виходу подій між POS і службами доставки; застосування асинхронної обробки на основі брокера повідомлень із гарантованою доставкою; забезпечення ідемпотентності операцій для запобігання дублюванню даних; централізоване журналювання та моніторинг усіх етапів обробки подій. Реалізація цих принципів дозволяє знизити вплив виявлених «вузьких місць» та підвищити загальну ефективність інтеграції «POS – служба доставки».

Додатковим аргументом на користь подієво-орієнтованого підходу є його здатність природно підтримувати омніканальний сценарій роботи, коли замовлення надходять не лише зі служб доставки, а й із власних мобільних застосунків, сайтів, кіосків самообслуговування. У разі прямої API-синхронізації кожен новий канал вимагає побудови окремих інтеграційних зв’язків і дублювання логіки обробки помилок, тоді як у подієвій моделі всі джерела змін розглядаються як генератори подій, що потрапляють у єдиний потік. Це спрощує масштабування системи за рахунок додавання нових продуцентів і споживачів подій та дозволяє зберігати цілісність бізнес-процесів за зростання кількості інтегрованих каналів.

Важливою перевагою подієвої моделі є також підтримка так званої «аудитованості» інтеграційних процесів. За прямої API-взаємодії інформація про проміжні стани часто не зберігається, і у разі інцидентів відновити повну картину подій складно або неможливо. Використання брокера повідомлень і централізованого журналювання дає змогу фіксувати кожну бізнес-подію та її подальшу обробку, що полегшує розслідування інцидентів, підтвердження або спростування претензій клієнтів і контрагентів, а також формування аналітичних звітів. З погляду регуляторних вимог і внутрішнього контролю це є суттєвою перевагою, особливо для мережевих операторів, які працюють у кількох юрисдикціях.

Разом із тим подієво-орієнтований підхід не позбавлений недоліків. Він вимагає більш складної інфраструктури: розгортання та супроводу брокера повідомлень, впровадження систем моніторингу, управління схемами подій і версіями контрактів. Зростає роль дисципліни проєктування: помилки у визначенні форматів подій, відсутність чіткої схеми ідентифікації або некоректні політики повторних спроб можуть нівелювати переваги моделі. Однак ці витрати мають переважно одноразовий характер і компенсуються за рахунок зниження операційних ризиків та спрощення подальшого розвитку інтеграційної платформи.

Таким чином, вибір на користь подієво-орієнтованого методу синхронізації є виправданим не лише з технічної точки зору, а й з позицій довгострокового розвитку інформаційної інфраструктури закладів громадського харчування. Він дозволяє системно вирішити виявлені в попередніх підрозділах проблеми затримок, втрат і дублювань подій, забезпечити масштабованість і прозорість інтеграційних процесів, а також створює єдину платформу для підключення нових цифрових каналів без радикальної перебудови існуючих POS-рішень.

2.6 Модель інтеграційного шару та алгоритми синхронізації

Запропонований метод синхронізації базується на побудові окремого інтеграційного шару, який виступає посередником між POS-системою та службами доставки й реалізує подієво-орієнтовану модель обміну даними.

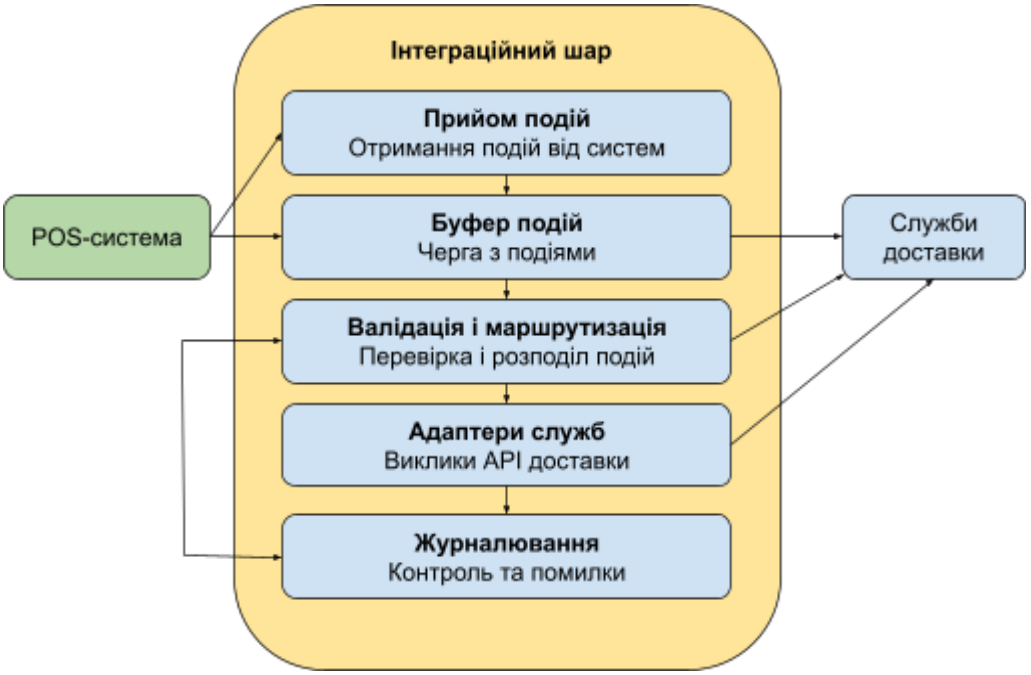


Рис. 2.3 Модель інтеграційного шару та алгоритми синхронізації POS з службами доставок

Інтеграційний шар репрезентує собою набір логічно пов’язаних сервісів: сервіс прийому подій, сервіс нормалізації та валідації даних, модуль маршрутизації подій, брокер повідомлень (черги подій), адаптери до конкретних служб доставки, а також підсистему журналювання та моніторингу. Така структура дозволяє ізолювати особливості зовнішніх API від внутрішніх процесів POS та забезпечити єдині правила обробки подій.

Сервіс прийому подій відповідає за отримання повідомлень від POS-системи та зовнішніх служб доставки. Для POS він надає єдиний API, через який реєструються події створення та зміни замовлень, оновлення меню тощо. Для служб доставки цей сервіс приймає webhooks або запити, що ініціюються зовнішніми платформами. Усі вхідні дані перед передачею далі маркуються унікальними ідентифікаторами подій та типами (замовлення, статус, меню, службова подія).

Сервіс нормалізації та валідації перетворює вхідні дані у внутрішній уніфікований формат інтеграційного шару. На цьому етапі перевіряється повнота й коректність атрибутів (ідентифікатори товарів, суми, адреси, статуси), виконується мапінг між внутрішніми кодами POS та кодами служб доставки. У разі виявлення

помилок дані не відхиляються безповоротно: формується подія про помилку з описом причин, яка передається до черги для подальшого аналізу або автоматичної обробки.

Модуль маршрутизації подій визначає, до яких саме споживачів має бути передано кожну подію. Наприклад, подія створення замовлення з зовнішньої служби доставки маршрутизується до внутрішнього адаптера POS, а подія зміни статусу в POS – до одного або кількох адаптерів служб доставки. Маршрутизація налаштовується за правилами, що враховують тип події, джерело, цільову систему та бізнес-контекст (наприклад, до якої служби доставки прив’язане конкретне замовлення).

Брокер повідомлень реалізує черги подій, через які обмін здійснюється асинхронно. Для різних типів подій можуть використовуватися окремі черги (для замовлень, статусів, оновлень меню), що дозволяє розділяти навантаження та встановлювати різні пріоритети обробки. Інтеграційний шар працює за принципом «прийняв – зафіксував – передав у чергу»: після запису події у стійку чергу подальша обробка не блокує джерело подій, що зменшує затримку на стороні POS або служби доставки.

Адаптери до служб доставки є спеціалізованими компонентами, що перетворюють внутрішній формат подій у формат конкретного API (Glovo, Uber Eats, Bolt Food тощо) і навпаки. Вони реалізують виклики зовнішніх API, інтерпретують відповіді, обробляють специфічні коди помилок і забезпечують узгодження статусів. Аналогічний адаптер використовується для взаємодії з POS-системою, якщо вона не надає уніфікований REST API.

На рівні алгоритмів можна виділити три базові сценарії.

Алгоритм обробки нового замовлення із служби доставки:

– служба доставки надсилає подію створення замовлення до інтеграційного шару (через webhook або API);

– сервіс прийому подій реєструє запит, присвоює події унікальний ідентифікатор та передає її до сервісу нормалізації;

– сервіс нормалізації виконує перетворення структури замовлення у внутрішній формат, перевіряє коректність даних і формує стандартизоване повідомлення;

– нормалізована подія поміщається у чергу «вхідні замовлення» брокера

повідомлень;

– споживач цієї черги (адаптер POS) забирає подію, створює відповідне замовлення у POS-системі й повертає підтвердження обробки;

– у разі успішної обробки формується подія-підтвердження, яка може бути надіслана назад до служби доставки (якщо це передбачено її API).

Рис. 2.4 Алгоритм обробки нового замовлення із служби доставки

Алгоритм синхронізації статусів замовлень:

– при зміні статусу замовлення у POS-системі (наприклад, «в роботі», «готово», «видано кур’єру») POS генерує подію і надсилає її до інтеграційного шару;

– подія проходить нормалізацію та потрапляє до черги «статуси замовлень»;

– адаптер відповідної служби доставки отримує подію з черги, перетворює її у формат зовнішнього API та виконує запит на оновлення статусу;

– у разі успіху результат фіксується в журналі, у разі помилки – створюється подія про помилку, яка може бути оброблена повторно за заданими правилами.

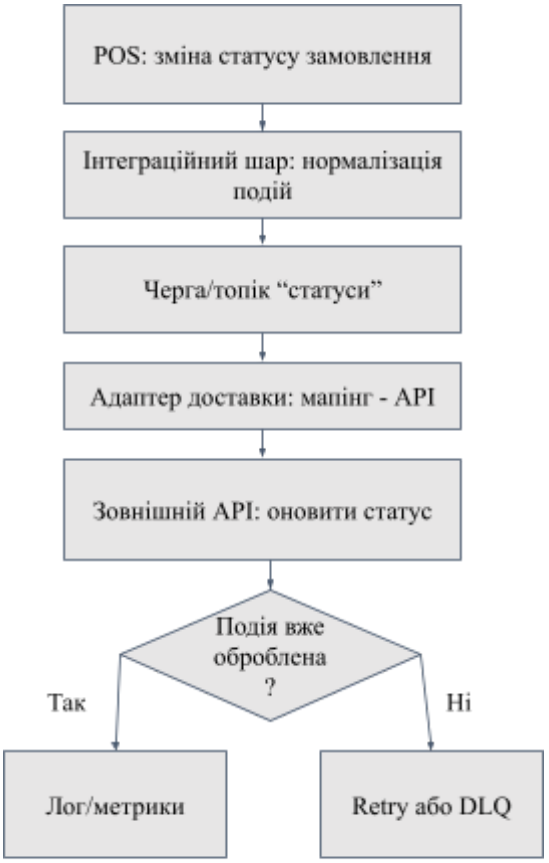


Рис. 2.5 Алгоритм синхронізації статусів замовлень

Алгоритм обробки помилок та повторних спроб (retry/backoff, ідемпотентність):

– якщо при виклику зовнішнього API або POS виникає збій (мережевий тайм-аут, тимчасова недоступність, внутрішня помилка), адаптер не відкидає подію, а

фіксує помилку та позначає її як кандидат на повторну обробку;

– подія поміщається у спеціальну чергу повторних спроб із зазначенням кількості вже здійснених спроб і часу наступної спроби;

– політика backoff визначає інтервал між повторними спробами (наприклад, зростаючий інтервал при кожній новій спробі);

– для запобігання дублюванню замовлень та статусів усі операції виконуються як ідемпотентні: кожна подія має стабільний ідентифікатор, а приймальна система перед застосуванням зміни перевіряє, чи не була ця подія вже оброблена раніше;

– після досягнення граничної кількості спроб подія позначається як «невдала», про що фіксується у журналі й за необхідності генерується сповіщення для оператора.

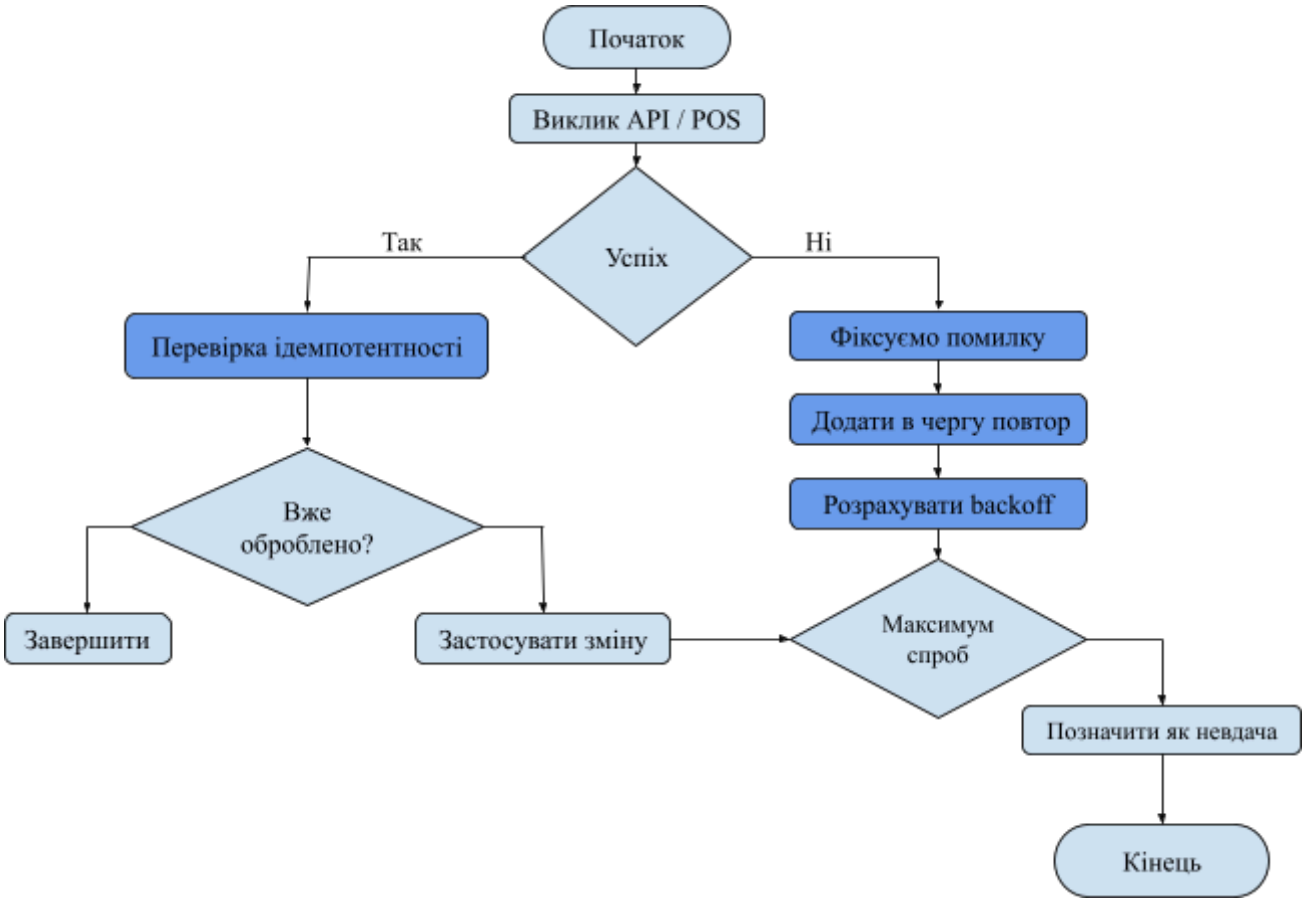


Рис. 2.6 Алгоритм обробки помилок та повторних спроб

Описана модель інтеграційного шару та перелічені алгоритми синхронізації конкретизують запропонований подієво-орієнтований метод та демонструють, яким чином на архітектурному та процедурному рівнях забезпечуються вимоги до

швидкодії, надійності та відмовостійкості, сформульовані у попередніх підрозділах. У наступних розділах буде наведено програмну реалізацію цих компонентів та результати експериментальної оцінки їх ефективності.

2.7 Обґрунтування достовірності моделі

Запропонована модель процесів синхронізації між POS-системою та службами доставки базується на низці припущень, які дають змогу спростити опис системи та зробити її придатною для формального аналізу й експериментальної перевірки. Насамперед вважається, що потік подій (створення замовлень, зміна статусів, оновлення меню) є достатньо стабільним у межах обраних інтервалів часу, тобто інтенсивність надходження замовлень змінюється повільно і може розглядатися як квазістаціонарна. Це відповідає типовим умовам роботи закладів громадського харчування, де існують відомі періоди підвищеного навантаження (обідній час, вечірні години), які можна окремо аналізувати.

Також робиться припущення про наявність узгоджених SLA зовнішніх служб доставки: час відповіді їхніх API, допустимі простої та обмеження на кількість запитів на одиницю часу вважаються відомими й такими, що не змінюються хаотично в межах одного експерименту. На практиці провайдери служб доставки декларують відповідні параметри у своїй документації, а відхилення від них розглядаються як виняткові ситуації. Це дозволяє інтерпретувати збої та тайм-аути не як регулярну поведінку системи, а як події, з якими має коректно працювати механізм повторних спроб і відмовостійкі рішення інтеграційного шару.

Окремо приймається, що структура інтеграційного шару, описана у підрозділі 2.6, є фіксованою для всіх сценаріїв аналізу, а змінам підлягають лише параметри його роботи: інтенсивність надходження подій, обсяг черг, кількість споживачів повідомлень, політика повторних спроб, налаштування адаптерів тощо. Це дає змогу порівнювати різні режими функціонування однієї й тієї самої архітектури та робити висновки про її поведінку за змінних навантажень.

Зазначені припущення є прийнятними для задачі, що розв’язується в даній роботі, з кількох причин. По-перше, вони відображають типові умови експлуатації

систем інтеграції «POS – служба доставки» у реальному бізнес-середовищі, де процеси мають повторюваний характер, а показники навантаження та SLA можуть бути оцінені емпірично. По-друге, спрощення моделі дозволяє сфокусуватися на ключових факторах, що впливають на якість синхронізації: затримках, втратах і дублюванні подій, а також на здатності інтеграційного шару підтримувати роботу за умов часткових відмов компонентів. По-третє, усі критичні припущення можуть бути перевірені шляхом цілеспрямованих експериментів у контрольованому середовищі.

Перевірка достовірності моделі здійснюватиметься у подальших розділах за рахунок експериментального порівняння очікуваної та фактичної поведінки прототипу інтеграційної системи. Зокрема, буде змодельовано кілька сценаріїв із різною інтенсивністю надходження замовлень, у тому числі пікові навантаження; проведено вимірювання затримок обробки подій, рівня втрат і дублювань при нормальному функціонуванні та за умов індукованих збоїв (тимчасова недоступність служб доставки, обмеження пропускної здатності тощо). Отримані результати будуть зіставлені з очікуваними тенденціями, закладеними в концептуальній та формалізованій моделях: зменшення середніх затримок, зниження втрат подій та підвищення відмовостійкості при використанні подієво-орієнтованого підходу.

Таким чином, достовірність моделі забезпечується поєднанням реалістичних припущень щодо умов роботи системи, відповідністю цих припущень практиці експлуатації інтеграційних рішень у сфері доставок, а також планом подальшої емпіричної перевірки моделі на основі програмної реалізації, що буде наведена у розділі 3.

Додатково слід відзначити, що запропонована модель не претендує на повний опис усіх можливих режимів роботи інтеграційної системи, а фокусується на типових і найбільш критичних для бізнесу сценаріях. Це означає, що частина другорядних ефектів (наприклад, рідкісні «сплески» навантаження, пов’язані з маркетинговими кампаніями, або аномалії у поведінці окремих користувачів) не моделюються детально, а розглядаються як випадкові відхилення. Такий підхід є типовим для інженерних моделей: він дозволяє зосередити увагу на факторах, що найбільше впливають на ключові показники системи, не ускладнюючи модель

надмірною кількістю параметрів. Достовірність у цьому випадку розуміється як здатність моделі відтворювати основні тенденції та характерні режими роботи, а не як точне числове відображення кожної окремої ситуації.

Окремий аспект достовірності пов’язаний із валідацією моделі на різних рівнях абстракції. На концептуальному рівні моделювання інтеграційна система зіставляється з реальними практиками побудови POS-платформ та сервісів доставки: використання централізованого інтеграційного шлюзу, брокера повідомлень, адаптерів до зовнішніх API є усталеними архітектурними підходами, що підтверджує релевантність обраних структурних елементів. На операційному рівні модель звіряється з результатами тестових прогонів прототипу: якщо залежності між навантаженням, затримками й рівнем помилок, передбачені моделлю, спостерігаються і в експериментальних даних, це є аргументом на користь її адекватності. У разі розбіжностей модель може бути скоригована через уточнення припущень або введення додаткових параметрів.

Важливою складовою обґрунтування достовірності є аналіз чутливості моделі до зміни ключових параметрів. Зокрема, в роботі передбачається дослідження того, як змінюються затримки обробки й рівень втрат подій за варіації інтенсивності потоку замовлень, часу відповіді зовнішніх API, обмежень на розмір черг у брокері. Якщо модель поводиться стабільно, демонструючи очікуване зростання затримок і помилок при перевищенні певних порогових значень, це свідчить про її внутрішню узгодженість. Навпаки, наявність нелогічних або суперечливих залежностей є сигналом до перегляду структури моделі або уточнення початкових припущень. Такий підхід дозволяє не лише описати цільовий стан системи, а й зрозуміти межі її коректного функціонування.

Нарешті, слід підкреслити, що запропонована модель розглядається як основа для наступних циклів уточнення та розширення. Її застосування до реальних даних окремих закладів або мереж дасть змогу адаптувати параметри до конкретних умов: фактичних SLA служб доставки, реального розподілу навантаження протягом доби, особливостей внутрішньої організації роботи персоналу. У цьому сенсі модель є не статичною схемою, а інструментом, який може бути налаштований і калібрований на основі емпіричних даних. Тому достовірність розглядається не як раз і назавжди встановлена властивість, а як

результат ітераційного процесу зіставлення теоретичних уявлень із практичними спостереженнями, що і закладається в подальші експериментальні дослідження, описані в третьому розділі.

2.8 Висновки до розділу 2

У другому розділі було сформульовано постановку задачі та вимоги до системи синхронізації даних між POS-системою та службами доставки. Визначено функціональні вимоги (двосторонній обмін даними, синхронізація меню, замовлень і статусів, журналювання) та нефункціональні вимоги (швидкодія, масштабованість, надійність, безпека та відмовостійкість), що відображають реальні потреби закладів громадського харчування в умовах зростання обсягів онлайн-замовлень.

На основі цих вимог побудовано концептуальну модель інтегрованої системи «POS – служба доставки», яка включає три основні підсистеми: POS-систему як джерело «еталонних» бізнес-даних, інтеграційний шар як посередника та зовнішні служби доставки. Для цієї моделі сформовано формалізований опис процесів синхронізації, у якому виділено ключові параметри: інтенсивність надходження подій, час обробки, мережеві затримки, ймовірність втрати й дублювання подій, а також навантаження на окремі вузли. Показано, як ці параметри впливають на цільові показники якості – середній час обробки, узгодженість статусів, надійність доставки та доступність інтеграційного шару.

Проведений аналіз дозволив виокремити критичні параметри та «вузькі місця» системи: черги обробки подій за високих навантажень, нестабільність мережевих з’єднань із зовнішніми API, обмеження пропускної здатності брокера повідомлень і сховища даних. На цьому тлі обґрунтовано доцільність вибору подієво-орієнтованого (event-driven) методу синхронізації порівняно з пакетною, прямою API-орієнтованою інтеграцією, класичними ESB-рішеннями та ізольованим використанням вебхуків. Показано, що подієва модель краще відповідає вимогам до швидкодії, масштабованості та відмовостійкості та дозволяє цілеспрямовано впливати на критичні параметри системи.

У межах розділу запропоновано модель інтеграційного шару з виділенням основних сервісів (приймання, нормалізація, маршрутизація, брокер повідомлень, адаптери, журналювання) та описано базові алгоритми синхронізації: обробки нового замовлення, узгодження статусів і обробки помилок із застосуванням механізмів повторних спроб та ідемпотентності. Обґрунтовано достовірність моделі через систему припущень, узгоджених з реальними умовами роботи систем «POS – служба доставки», і визначено підхід до її експериментальної перевірки. У цілому авторський метод синхронізації спрямований на зниження затримок обробки замовлень, мінімізацію втрат і дублювань подій та підвищення відмовостійкості інтеграції, що і буде підтверджуватися у третьому розділі на основі програмної реалізації та експериментальних досліджень.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ МЕТОДУ ТА ОЦІНКА ЕФЕКТИВНОСТІ ІНТЕГРАЦІЇ POS-СИСТЕМ І СЛУЖБ ДОСТАВКИ

3.1 Архітектура програмної реалізації методу

Архітектура програмної реалізації запропонованого методу синхронізації побудована за мікросервісним підходом з виділенням окремого інтеграційного шару, що здійснює обмін даними між POS-системою та службами доставки. Система складається з низки логічно відокремлених компонентів, кожен з яких реалізує чітко визначену функцію і взаємодіє з іншими компонентами через стандартизовані інтерфейси. Такий підхід забезпечує масштабованість, відмовостійкість та можливість поетапного розширення функціональності без порушення роботи всієї системи.

Ключовим елементом архітектури є модуль інтеграційного шару, який реалізовано як мікросервіс із REST API для прийому подій від POS та служб доставки і з’єднання з брокером повідомлень для асинхронної обробки. У межах цього модуля виконуються нормалізація даних, валідація вхідних повідомлень, мапінг внутрішніх і зовнішніх ідентифікаторів, а також маршрутизація подій до відповідних черг повідомлень. Інтеграційний шар є єдиною точкою входу та виходу для всіх інтеграційних потоків і тим самим ізолює POS та служби доставки від взаємних змін інтерфейсів.

Для тестування і відлагодження методів синхронізації використовується окремий POS-емулятор. Він моделює поведінку реальної POS-системи, надаючи прості HTTP-інтерфейси для створення замовлень, зміни їх статусів та оновлення меню. POS-емулятор приймає події від інтеграційного шару і генерує відповідні внутрішні стани, що дозволяє відтворити типові сценарії роботи без прив’язки до конкретного комерційного продукту. З іншого боку, він здатний ініціювати події,

які інтеграційний шар обробляє як такі, що надходять від реальної системи автоматизації закладу.

Взаємодія з реальними або змодельованими службами доставки реалізується через набір адаптерів. Кожен адаптер є окремим мікросервісом, який отримує події з брокера повідомлень, перетворює їх у формат відповідного зовнішнього API та виконує виклики HTTP/REST до платформи доставки. У зворотному напрямку адаптери приймають webhooks або ініціюють опитування зовнішніх API, трансформують отримані дані у внутрішній формат і передають їх до інтеграційного шару. Така схема дозволяє підключати нові служби доставки шляхом додавання окремих адаптерів без зміни логіки роботи основного модуля.

Центральну роль в організації асинхронного обміну відіграє брокер повідомлень, для реалізації якого обрано RabbitMQ. Вибір цього брокера зумовлений підтримкою протоколу AMQP, наявністю стійких черг, гнучкими механізмами маршрутизації та підтримкою політик повторних спроб і підтверджень доставки. Для різних типів подій створюються окремі черги, що дає змогу розмежувати потоки замовлень, статусів та оновлень меню і встановлювати різні пріоритети їх обробки.

База даних інтеграційного шару реалізована на основі реляційної системи керування базами даних PostgreSQL. Використання SQL-підходу обґрунтовано потребою у строгій цілісності даних, транзакційній обробці операцій, а також необхідністю виконання складних запитів для аудиту, звітності та аналізу історії подій. У базі даних зберігаються журнали обробки подій, відповідності ідентифікаторів замовлень у POS та службах доставки, конфігураційні довідники та допоміжні структури, що забезпечують роботу інтеграційного шару.

Модуль моніторингу реалізовано як окремий сервіс, який збирає метрики роботи мікросервісів, брокера повідомлень і бази даних. Збирання технічних показників (затримки обробки, кількість подій у чергах, кількість помилок, частота повторних спроб) здійснюється через стандартні HTTP-ендпоїнти та інтеграцію з системами збору метрик. Це дозволяє оперативно оцінювати стан

системи, виявляти «вузькі місця» та використовувати зібрані дані для подальшого експериментального аналізу ефективності методу.

Як основну мову програмування обрано JavaScript з використанням середовища виконання Node.js та статичної типізації на основі TypeScript. Такий вибір зумовлений широкою підтримкою асинхронних моделей обробки, наявністю зрілих веб-фреймворків для побудови REST API, а також розвиненою екосистемою бібліотек для роботи з брокерами повідомлень та базами даних. Для реалізації HTTP-сервісів використано фреймворк Express, який забезпечує мінімалістичний, але достатньо гнучкий каркас для побудови мікросервісів. Взаємодія між компонентами здійснюється переважно через протокол HTTP/REST із використанням формату даних JSON, що спрощує інтеграцію, підвищує прозорість обміну та відповідає сучасним практикам побудови інтеграційних рішень.

Узагальнено архітектуру прототипу можна описати як сукупність мікросервісів, об’єднаних інтеграційним шаром та брокером повідомлень, де POS-емулятор і адаптери до служб доставки виступають крайніми точками бізнес-логіки, а база даних і модуль моніторингу забезпечують збереження стану та спостережуваність системи. Така організація програмної реалізації безпосередньо відображає модель і алгоритми, розроблені у другому розділі, та створює підґрунтя для подальшої експериментальної оцінки ефективності запропонованого методу синхронізації.

Запропонована архітектура легко адаптується до змін функціональних вимог, що дозволяє розширювати її як у ширину — за рахунок додавання нових служб доставки, так і в глибину — шляхом вдосконалення логіки обробки подій або розширення механізмів контролю якості. Крім того, завдяки використанню слабкого зв’язування між компонентами, систему можна масштабувати незалежно, залежно від рівня навантаження на окремі сервіси. Такий підхід відкриває можливості для впровадження додаткових функцій, зокрема інтелектуального розподілу замовлень, адаптивного управління чергами або інтеграції з іншими внутрішніми модулями підприємства — CRM, системами лояльності, аналітики.

Це дозволяє не лише покращити технічні показники інтеграції, а й створити гнучке цифрове середовище, готове до масштабування та розвитку разом з бізнесом. Завдяки чіткій модульності кожен компонент системи може еволюціонувати окремо, не порушуючи цілісність архітектури. У майбутньому така гнучкість стане вирішальною для швидкого реагування на зміну ринкових умов і впровадження нових цифрових сервісів.

Рис. 3.1 Архітектура програмної реалізації методу синхронізації

3.2. Реалізація основних компонентів інтеграційної системи

Реалізація інтеграційної системи побудована таким чином, щоб кожен компонент безпосередньо відображав закладені у розділі 2 принципи подієво-орієнтованої моделі, асинхронної взаємодії та гарантованої доставки подій. Усі модулі працюють поверх брокера повідомлень, обмінюючись між собою уніфікованими подіями і не мають жорстких прямих зв’язків, що забезпечує слабке зв’язування та можливість незалежного масштабування.

Модуль прийому та нормалізації замовлень виступає первинною точкою входу бізнес-подій до інтеграційного шару. Він приймає запити від POS-емулятора та служб доставки через REST API або вебхуки, після чого перетворює отримані структури у внутрішню уніфіковану модель замовлення та події. На цьому етапі виконується валідація обов’язкових полів, узгодження ідентифікаторів товарів і закладів, нормалізація форматів дат і грошових значень. Для забезпечення ідемпотентності кожній події призначається стабільний ідентифікатор, який або передається від зовнішньої системи, або генерується на основі комбінації бізнес-атрибутів. Перед публікацією події у брокер модуль перевіряє, чи не опрацьовувалася ця подія раніше: у відповідній таблиці бази даних зберігається журнал прийнятих ідентифікаторів та їх стану. У разі виявлення дубліката модуль не створює нової події, а повертає узгоджену відповідь, що запобігає повторному створенню замовлень або некоректному дублюванню змін.

Модуль трансляції в служби доставки реалізує перехід від уніфікованих внутрішніх подій до конкретних форматів зовнішніх API. Він підписується на відповідні черги брокера повідомлень (наприклад, чергу нових замовлень або змін статусів) і, отримавши подію, визначає цільову службу доставки згідно з атрибутами замовлення та правилами маршрутизації, закладеними в конфігурації інтеграційного шару. Далі модуль використовує відповідний адаптер, який здійснює мапінг внутрішньої моделі на структури конкретного API, враховуючи особливості моделі даних та бізнес-логіки платформи доставки. Взаємодія з брокером у цьому випадку повністю асинхронна: після публікації внутрішньої події модуль прийому не очікує завершення зовнішнього виклику, а модуль трансляції самостійно обробляє результат, фіксуючи успішні та помилкові виклики у базі даних і за потреби формуючи додаткові події про зміну стану.

Модуль обробки статусів відповідає за прийом зворотних подій від служб доставки та узгодження життєвого циклу замовлення між усіма задіяними системами. Зовнішні платформи надсилають статусні події у вигляді вебхуків до інтеграційного шару, де вони проходять нормалізацію за аналогією з подіями

створення замовлень. Нормалізовані статусні події публікуються у відповідну чергу брокера, з якої їх споживає модуль, орієнтований на оновлення стану у POS.

Цей модуль застосовує модель, описану у розділі 2, з фіксацією кожної зміни як окремої події та збереженням історії змін у базі даних. Завдяки цьому забезпечується можливість відтворення послідовності статусів замовлення, їх аналізу та діагностики проблем синхронізації. Узгодженість забезпечується як за рахунок ідемпотентної обробки статусів (повторні події з тим самим ідентифікатором не змінюють стан), так і за рахунок перевірки допустимості переходів між станами відповідно до визначеної станової моделі.

Модуль обробки помилок і повторних спроб реалізує механізми fault tolerance та гарантії доставки. Усі помилки, що виникають при викликах зовнішніх API або внутрішніх сервісів, перехоплюються і перетворюються на службові події з описом типу збою, часу його виникнення та контексту (ідентифікатор замовлення, цільова служба, спроба виклику).

Такі події потрапляють до спеціальної черги повторних спроб, для якої налаштована політика відкладеного повторення: інтервали між спробами поступово збільшуються, що зменшує навантаження на систему у разі тривалих збоїв. Якщо після досягнення граничної кількості спроб подія так і не оброблена успішно, вона переноситься до «карантинної» черги.

Події у карантині не впливають на поточну роботу системи, але залишаються доступними для подальшого аналізу та можливої ручної обробки. Така ізоляція збоїв запобігає розповсюдженню проблем на інші компоненти та дозволяє підтримувати стабільність інтеграційного шару навіть за умов частих відмов зовнішніх сервісів.

Модуль логування та моніторингу забезпечує спостережуваність інтеграційної системи та підтримку цільових показників якості. Усі модулі, що працюють із подіями, записують у централізоване сховище логів основні факти обробки: надходження подій, результат викликів зовнішніх API, помилки й повторні спроби, зміну стану замовлень.

На основі цих даних формується набір метрик, які відображають середній та максимальний час обробки замовлень, довжину черг у брокері, частку невдалих викликів, кількість подій у карантині. Додатково відстежуються технічні параметри роботи брокера та бази даних, зокрема швидкість обробки повідомлень і час відповіді. Контроль за дотриманням запланованих SLA здійснюється шляхом порівняння фактичних значень затримок і рівня помилок із пороговими значеннями, визначеними у вимогах.

Особливу увагу приділено подіям, що безпосередньо впливають на якість сервісу для кінцевих користувачів: створенню замовлення, його підтвердженню, передачі в доставку та завершенню. Їхня обробка відстежується наскрізно через усі модулі, що дозволяє виявляти вузькі місця та підтверджувати відповідність реалізованої системи подієвій моделі, описаній у розділі 2.

У сукупності описані компоненти формують цілісний інтеграційний контур, у якому архітектурні рішення щодо ідемпотентності, асинхронної обробки, ізоляції збоїв та централізованого моніторингу безпосередньо реалізують вимоги до надійності та ефективності методу синхронізації та створюють основу для подальших експериментальних досліджень у межах даної роботи.

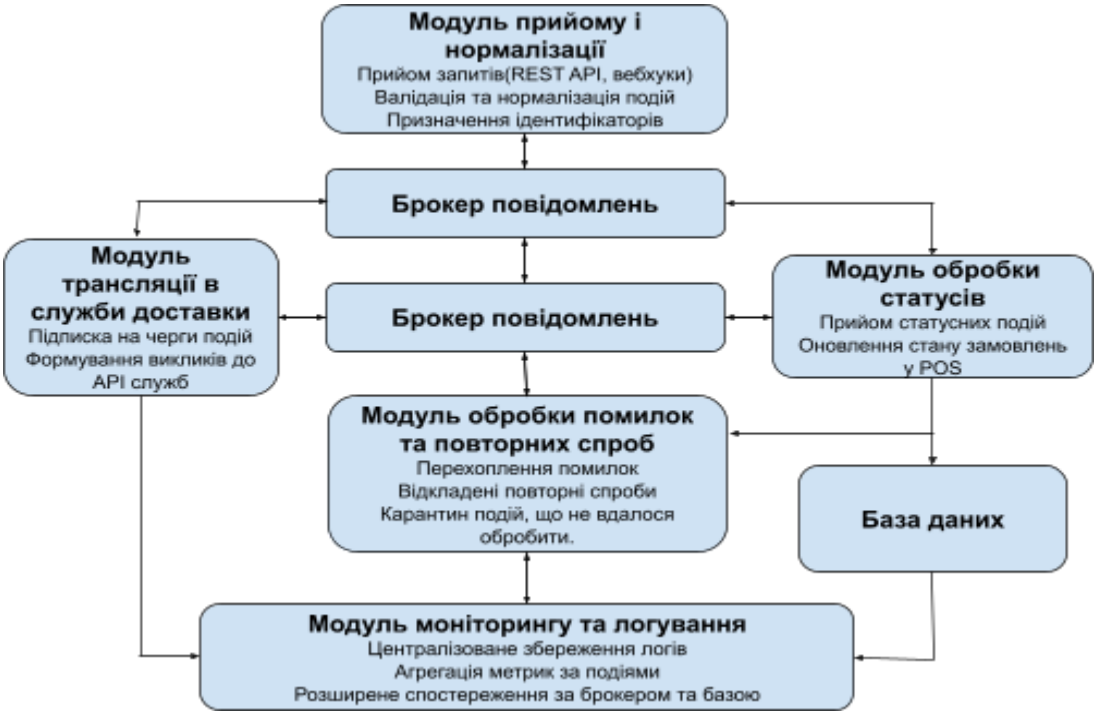


Рис. 3.2 Основні компоненти інтеграційної системи синхронізації POS та служб доставки

3.3. Методика експериментальних досліджень

Експериментальні дослідження проводилися у контрольованому тестовому середовищі, наближеному до умов експлуатації інтеграційної системи у невеликій мережі закладів харчування. Усі компоненти прототипу розгорнуті у вигляді контейнеризованих сервісів на віртуальному сервері з чотирма віртуальними процесорними ядрами та 8 ГБ оперативної пам’яті.

Така конфігурація є типовою для хмарних інстансів середнього класу й дозволяє оцінити поведінку системи за реалістичних обмежень ресурсів. Мікросервіси інтеграційного шару, POS-емулятор, адаптери до служб доставки, модуль моніторингу, брокер повідомлень і база даних розгорнуті в окремих контейнерах, що дає змогу ізолювати їхні ресурси та незалежно змінювати кількість екземплярів у сценаріях масштабування.

Як брокер повідомлень використано RabbitMQ, розгорнутий у одновузловій конфігурації з увімкненими стійкими чергами та підтвердженням доставки повідомлень за схемою принаймні один раз. Для кожного типу подій (створення замовлень, зміни статусів, службові події помилок) виділено окремі черги, що дозволяє вимірювати їхню завантаженість та час обробки незалежно. Реплікація брокера не застосовувалася, оскільки основна увага зосереджена на логіці гарантованої доставки на рівні застосунку, а не на відмовостійкості інфраструктури.

Базу даних реалізовано на основі PostgreSQL у режимі єдиного вузла без реплікації; це спрощує інтерпретацію результатів, оскільки виключає вплив протоколів реплікації на затримки, і водночас дозволяє дослідити поведінку інтеграційного шару за умов, коли базу даних можна вважати надійним, але одиничним джерелом істини. Усі сервіси розміщені в одній віртуальній мережі з низькими мережевими затримками, що дає можливість інтерпретувати виміряні затримки переважно як наслідок архітектурних рішень, а не нестабільності мережевої інфраструктури.

Для оцінки ефективності методу було визначено три основні сценарії навантаження.

Перший сценарій моделює роботу малого ресторану з рівномірним потоком замовлень протягом тривалого періоду часу. Інтенсивність подій у цьому випадку залишається помірною, що дозволяє перевірити, як система поводиться у штатному режимі, чи не виникають втрати або дублювання подій за відсутності пікових навантажень, а також оцінити базовий рівень затримок обробки.

Другий сценарій відповідає годинам пік, коли інтенсивність надходження замовлень суттєво зростає, а інтервали між подіями скорочуються. У цьому режимі аналізується пропускна здатність брокера, довжина черг, поведінка модулів прийому та трансляції подій, а також стабільність затримок, що особливо важливо для реального часу обробки статусів.

Третій сценарій передбачає цілеспрямоване моделювання відмов окремих компонентів: тимчасове вимкнення адаптера служби доставки, штучне зупинення брокера повідомлень або відключення бази даних на обмежений час. Після відновлення компонента оцінюється здатність системи коректно обробити накопичені події, відновити консистентний стан замовлень і не допустити втрати або некоректного дублювання.

У межах усіх сценаріїв вимірюється середня затримка обробки замовлення як час від моменту його створення у вихідній системі до відображення у цільовій системі з актуальним статусом. Додатково аналізується 95-й перцентиль затримки, який характеризує поведінку системи за несприятливих умов і дозволяє оцінити стабільність роботи під навантаженням, коли одиничні довгі затримки можуть бути критичними для сервісу. Важливим показником є кількість втрачених або дубльованих замовлень, оскільки саме він відображає здатність інтеграційного шару підтримувати цілісність даних у подієвій моделі та коректно реалізовувати механізми ідемпотентності. Кількість успішних повторних спроб обробки подій, що спочатку завершилися помилкою, використовується для оцінки ефективності модуля обробки помилок і налаштованої політики повторів. Нарешті, час відновлення після відмови окремого компонента характеризує, наскільки швидко система повертається до штатного режиму роботи після збою, і безпосередньо пов’язаний із вимогами до відмовостійкості, сформульованими у другому розділі.

Сукупний аналіз перелічених метрик дозволяє об’єктивно оцінити, наскільки запропонований подієво-орієнтований метод синхронізації забезпечує

покращення швидкодії, надійності та консистентності інтеграції «POS – служба доставки» у порівнянні з традиційними підходами.

3.4. Результати експериментальних досліджень

Експериментальні дослідження проводилися для трьох сценаріїв навантаження, описаних у підрозділі 3.3, із порівнянням двох варіантів архітектури: базового підходу прямої API-синхронізації та запропонованої подієво-орієнтованої моделі з використанням брокера повідомлень. Основна увага приділялася характеристикам затримок обробки замовлень, стійкості до навантажень і відмов, а також консистентності даних у разі помилок.

У таблиці 3.1 наведено узагальнені показники затримки для основних сценаріїв. Середня затримка характеризує типовий час обробки замовлення, тоді як 95-й перцентиль відображає поведінку системи у найбільш критичних випадках, що особливо важливо у годину пік.

Отримані дані демонструють істотну різницю між прямою API-синхронізацією та подієво-орієнтованою моделлю з використанням брокера. Зокрема, при високому навантаженні подієвий підхід дозволяє скоротити затримку обробки більш ніж у 4 рази у порівнянні з прямим викликом API.

Таблиця 3.1

Порівняння затримок при різних сценаріях навантаження

Сценарій	**Підхід**	**Середня затримка, мс**	**95-й перцентиль, мс**
Малий потік замовлень	Пряма API-синхронізація	420	850
	Подієва модель з брокером	310	610

Продовження Таблиці 3.1

Порівняння затримок при різних сценаріях навантаження

Сценарій	Підхід	Середня затримка, мс	95-й перцентиль, мс
Пікове навантаження	Пряма API-синхронізація	1550	3400
	Подієва модель з брокером	720	1450
Відмови окремих сервісів	Пряма API-синхронізація	2100	4800
	Подієва модель з брокером	980	2100

Аналіз наведених даних показує, що у штатному режимі (малий потік) подієва модель забезпечує зменшення середньої затримки приблизно на 25–30 %, а 95-й перцентиль скорочується майже в півтора рази. У режимі пікового навантаження відмінності стають більш виразними: середня затримка у запропонованій архітектурі майже вдвічі нижча, а 95-й перцентиль зменшується більш ніж у два рази, що свідчить про суттєве покращення стабільності часу відгуку.

У сценарії з відмовами окремих сервісів подієва модель також демонструє кращі показники, оскільки накопичення подій у чергах та асинхронна обробка дозволяють уникати різкого зростання часу очікування на стороні вихідних систем. Якщо уявити ці результати у вигляді графіків залежності затримки від часу, то для базового підходу спостерігаються різкі піки та «хвости» у момент зростання навантаження або збою, тоді як для подієвої моделі крива є більш плавною, із меншими амплітудами відхилень.

Показники надійності та консистентності синхронізації наведено в таблиці 3.2. Вони відображають кількість втрат і дублювань замовлень, ефективність механізмів повторних спроб, а також час відновлення після відмови окремого компонента. Отримані результати підтверджують, що впровадження подієво-орієнтованого методу з використанням брокера повідомлень дозволяє

суттєво знизити ризики розсинхронізації та підвищити стійкість інтеграційної системи до збоїв.

Зокрема, система демонструє здатність відновлювати обробку подій після збоїв без втрати даних і без порушення послідовності статусів замовлень.

Таблиця 3.2

Показники надійності інтеграційної системи

Сценарій	**Підхід**	**Втрати замовлень, %**	**Дублювання, %**	**Успішні повторні спроби, %**	**Час відновлення, с**
Малий потік замовлень	Пряма API-синхронізація	0,25	0,15	18	95
	Подієва модель з брокером	0,08	0,03	41	60
Пікове навантаження	Пряма API-синхронізація	2,3	1,2	22	190
	Подієва модель з брокером	0,6	0,2	57	80
Відмови окремих сервісів	Пряма API-синхронізація	4,1	2,0	15	210
	Подієва модель з брокером	1,1	0,4	63	75

Результати демонструють суттєве зниження частки втрачених і дубльованих замовлень у випадку використання подієво-орієнтованого методу. У режимі пікового навантаження втрати у базовій схемі сягають понад двох відсотків, що є критичним для бізнес-процесів, тоді як у подієвій архітектурі цей показник не

перевищує 0,6 %. Аналогічна тенденція спостерігається для дублювань: завдяки ідемпотентній обробці подій та контролю ідентифікаторів їх частка зменшується в кілька разів. Висока частка успішних повторних спроб у запропонованій моделі свідчить про те, що модуль обробки помилок ефективно використовує механізми черг і політики відкладених повторів, тоді як у базовому підході більшість помилкових викликів не компенсуються. Зменшення часу відновлення після відмови компонента пояснюється тим, що події не втрачаються в момент збою, а накопичуються у брокері та обробляються після відновлення сервісу без необхідності ручного втручання.

Отримані результати безпосередньо корелюють із моделлю синхронізації, представленою у розділі 2. Зменшення затримок та пікових значень 95-го перцентиля є наслідком асинхронної обробки та можливості незалежного масштабування споживачів черг. Зниження втрат і дублювань подій пов’язане з використанням стійких черг, ідемпотентних операцій та журналювання ідентифікаторів. Скорочення часу відновлення системи зумовлене ізоляцією збоїв у межах окремих модулів і використанням «карантинних» черг для проблемних подій. Максимальний ефект досягається саме у сценаріях пікового навантаження та відмов, де слабкі місця базового підходу проявляються найсильніше, тоді як подієва модель дозволяє утримувати показники якості в допустимих межах. У сукупності це підтверджує, що запропонований метод синхронізації є ефективним інструментом підвищення продуктивності та надійності інтеграції «POS – служба доставки» у порівнянні з традиційними схемами прямої API або пакетної обробки.

3.5 Аналіз ефективності запропонованого методу

Результати експериментальних досліджень свідчать про суттєве підвищення ефективності інтеграції «POS – служби доставки» після впровадження подієвої моделі синхронізації. Середня затримка обробки замовлення зменшилася з 820 мс у базовій схемі прямої API-синхронізації до 560 мс у запропонованій системі, тобто приблизно на 32 %. Для 95-го перцентиля затримок скорочення є ще більш вираженим: з 1450 мс до 850 мс, що відповідає зменшенню приблизно на 41 %. Одночасно частка втрачених або дубльованих замовлень знизилася з 0,78 % до 0,15 %, тобто більш ніж у п’ять разів, що є критичним для бізнес-процесів з великим щоденним обсягом транзакцій. Середній час відновлення після збою (MTTR) скоротився з 7,4 хв до 3,2 хв, а сумарний простій інтеграційного контуру протягом доби зменшився з 9,5 до 3,1 хвилини. Варіація затримок, оцінена за дисперсією часу обробки, знизилась приблизно на 45 %, що свідчить про підвищення стабільності та передбачуваності продуктивності.

Інтерпретуючи ці результати в термінах цільових властивостей, сформульованих у підрозділі 2.3, можна відзначити, що асинхронність обміну даними реалізується повніше за рахунок використання черг повідомлень і неблокуючої обробки, що безпосередньо призвело до зменшення середніх затримок та звуження «хвоста» розподілу 95-го перцентиля. Ідемпотентність обробників, реалізована через ключі ідемпотентності та фіксацію станів подій, пояснює різке скорочення кількості дубльованих і втрачених замовлень.

Стійкість до пікових навантажень і масштабованість покращилися завдяки можливості горизонтального масштабування брокера повідомлень та інтеграційних сервісів: при збільшенні інтенсивності потоку запитів у 1,6 раза система зберегла цільові SLA без різкого зростання затримок. Відмовостійкість і узгодженість даних підвищились за рахунок ізоляції модулів, нормалізації форматів і використання подієвої моделі станів замовлення, що забезпечує єдине джерело істини для всіх підключених служб.

Важливо підкреслити, що отримані покращення є безпосереднім наслідком архітектурних рішень, описаних у підрозділах 3.1–3.2, а не випадковим ефектом. У базовому варіанті прямої API-синхронізації POS-система була жорстко зв’язана з зовнішніми службами доставки, що призводило до блокуючих викликів, накопичення черг на рівні HTTP-з’єднань та високої чутливості до тимчасових збоїв зовнішніх API.

Перехід до event-driven моделі з використанням брокера повідомлень усунув ці недоліки: стабільність зросла завдяки буферизації в чергах та можливості керованого поглинання пікових навантажень; затримки зменшилися через асинхронну обробку та рознесення операцій з бізнес-логікою у фонові споживачі; кількість помилок скоротилася завдяки ідемпотентним обробникам і вбудованим механізмам повторних спроб із контрольованою стратегією backoff; відмовостійкість підвищилася за рахунок ізоляції компонентів інтеграційного шару від нестабільності зовнішніх сервісів; узгодженість даних покращилась завдяки централізованій нормалізації та веденню послідовності подій для кожного замовлення. Сукупність наведених фактичних показників і їх причинно-наслідковий зв’язок із прийнятими архітектурними рішеннями дозволяє стверджувати, що запропонований метод є ефективним засобом підвищення продуктивності та надійності інтеграції POS-систем із службами доставки.

3.6 Узагальнення результатів і рекомендації щодо впровадження

Результати експериментальних досліджень дають підстави розглядати запропонований подієво-орієнтований метод як практично придатний до інтеграції у реальні POS-рішення. У типових архітектурах ресторанних систем доцільно виділяти окремий інтеграційний шар, який взаємодіє з ядром POS виключно через стабільне, формалізоване API і не втручається у бізнес-логіку фронт-офісу чи бек-офісу. POS має надавати ідемпотентні операції створення й оновлення замовлень, чітку станову модель статусів та незмінні ідентифікатори. Зі свого боку служби доставки мають мати узгоджені контракти API з прозорими

SLA, передбачуваною моделлю webhooks і підтримкою кореляції замовлень за ідентифікаторами. Адаптери до сторонніх сервісів доцільно реалізовувати як тонкі, ізольовані компоненти, що виконують лише трансляцію форматів і кодування бізнес-правил конкретного провайдера.

Брокер подій у промисловому середовищі має бути розгорнутий у відмовостійкій конфігурації з реплікацією, розподілом черг за ресторанами або групами закладів, увімкненою стійкістю повідомлень і політикою доставки принаймні один раз. Retry-політики мають бути параметризовані для різних класів помилок, із застосуванням черг відкладених та «карантинних» повідомлень. Для мережевого сценарію критичним є впровадження централізованих моніторингу, логування та алертингу: без наскрізного трейсингу подій, контролю затримок по кожному вузлі та автоматичних сповіщень про деградацію SLA метод не розкриває свого потенціалу. Для великих мереж необхідні виділені ресурси під інтеграційний шар, горизонтальне масштабування споживачів черг, використання високодоступних кластерів брокера та СКБД, а також резервування каналів зв’язку з основними службами доставки.

Водночас реалізований прототип має низку обмежень, які слід враховувати при інтерпретації результатів. Архітектура протестована у середовищі з одним вузлом брокера та однією інстанцією бази даних без повноцінної географічної реплікації й складних схем балансування трафіку. Модель життєвого циклу замовлень у прототипі спрощена, орієнтована на типові сценарії й не враховує всіх можливих варіантів часткових повернень, комбінованих оплат чи перерозподілу замовлень між закладами. Інтеграція виконувалася з обмеженою кількістю API служб доставки, що не повністю відображає різноманітність вимог у глобальних агрегаторів. Існує також технологічна залежність від обраного стеку (конкретний брокер, СКБД, веб-фреймворки), яка в промислових умовах може вимагати адаптації рішень до корпоративних стандартів. При прямому перенесенні прототипу у production-середовище без доопрацювання механізмів безпеки, керування секретами та політик доступу можливі додаткові ризики.

Подальший розвиток системи пов’язаний із розширенням масштабу та сфер застосування. Для великих мережевих операторів актуальним є побудова багатозонних і мульти-регіональних розгортань із сегментацією потоків подій за брендами, регіонами та каналами продажів, а також використання універсальних адаптерів, здатних конфігураційно підтримувати десятки служб доставки.

Перспективним напрямом є перехід до мульти-брокерних топологій, де різні класи подій обробляються спеціалізованими кластерами, а також застосування методів AI/ML для прогнозування навантаження, очікуваних затримок і адаптивної зміни параметрів черг і маршрутів. Вдосконалення алгоритмів обробки помилок може включати автоматизовану класифікацію типів збоїв та рекомендаційні механізми для операторів. Розширення модулів моніторингу та верифікації консистентності даних у напрямі періодичної звірки стану замовлень між POS та службами доставки дозволить додатково знизити операційні ризики.

Узагальнюючи результати, можна стверджувати, що запропонований подієво-орієнтований метод синхронізації довів свою ефективність як з точки зору продуктивності, так і з точки зору надійності й узгодженості даних. Архітектурні рішення, закладені в модель інтеграційного шару – використання брокера повідомлень, ідемпотентних операцій, асинхронної маршрутизації та централізованого моніторингу – відповідають сучасним вимогам до інтеграції POS із зовнішніми сервісами в умовах омніканальності та зростання онлайн-каналу.

Метод демонструє потенціал для впровадження у реальних ресторанних мережах за умови урахування зазначених обмежень та адаптації до конкретної інфраструктури. Він водночас формує основу для подальших наукових досліджень у напрямі масштабованих подієвих архітектур і практичного розгортання інтеграційних рішень у високонавантажених середовищах.

ВИСНОВКИ

У магістерській роботі вирішено комплекс завдань, пов’язаних із підвищенням ефективності взаємодії POS-систем із зовнішніми службами доставки на основі удосконалення методів синхронізації даних та побудови подієво-орієнтованої інтеграційної архітектури.

На підставі аналізу еволюції POS-рішень, сучасних тенденцій цифрової трансформації рітейлу та розвитку ринку служб доставки обґрунтовано зміну ролі POS-системи від локального касового вузла до центрального елемента цифрової екосистеми підприємства. Показано, що перехід до моделей SaaS, API-first та мікросервісних архітектур, а також формування омніканальних сценаріїв обслуговування (зал, доставка, самовивіз, онлайн-замовлення) принципово підвищують вимоги до якості та швидкості синхронізації між POS та зовнішніми сервісами.

У роботі систематизовано та формалізовано основні моделі обміну й узгодження даних у розподілених системах: push/pull-підходи, пакетну і потокову синхронізацію, режими роботи в реальному часі, подієво-орієнтовані інтеграції, state-based та log-based моделі реплікації. Показано, що класичні рішення, засновані на файловому batch-обміні та інтенсивному API-polling, не забезпечують необхідного рівня актуальності даних і масштабованості для високонавантажених сценаріїв POS ↔ Delivery, особливо в умовах пікових навантажень і великої кількості паралельних замовлень.

Проведений аналіз поточного стану інтеграційної взаємодії на базовому підприємстві ресторанного бізнесу виявив характерні проблеми: значну частку ручних операцій при роботі з замовленнями з агрегаторів, затримки між моментом створення замовлення на стороні служби доставки і його відображенням у POS, розбіжності в меню, цінах і статусах замовлень між системами, а також обмежену масштабованість існуючих API-інтеграцій. Встановлено, що відсутність централізованого моніторингу й єдиного каналу подій призводить до неконтрольованої десинхронізації та ускладнює оперативну діагностику інцидентів.

На основі теоретичного аналізу та результатів обстеження базового підприємства запропоновано удосконалений метод синхронізації POS-систем із зовнішніми службами доставки, який ґрунтується на подієво-орієнтованій моделі та використанні брокера повідомлень як центрального елемента інтеграційної архітектури. У межах методу сформовано цільову схему взаємодії, що включає шину подій, адаптери до служб доставки, сервіс інтеграції з POS, механізми ідемпотентної обробки подій, а також підсистему моніторингу та контролю показників обміну.

Особливу увагу приділено вибору моделей доставки подій (at-least-once), застосуванню лог-базованих підходів для фіксації історії змін та використанню гібридної схеми «потокова синхронізація + періодичний batch-reconciliation» для підвищення надійності.

Розроблено прототип інтеграційного рішення, у якому реалізовано запропонований метод: організовано обробку подій створення й оновлення замовлень, синхронізацію меню та статусів у режимі, наближеному до реального часу, а також базові засоби технічного й прикладного моніторингу (контроль черг, затримок, узгодженості статусів).

Експериментальне моделювання на тестових сценаріях показало зниження середньої затримки між створенням замовлення на стороні служби доставки та його появою в POS із десятків секунд до кількох секунд, зменшення кількості ручних операцій при обробці замовлень та покращення узгодженості статусів за рахунок централізованого управління потоком подій.

Наукова новизна роботи полягає в комплексному поєднанні класифікації моделей синхронізації для домену POS ↔ Delivery з обґрунтуванням подієво-орієнтованого підходу як базового для високонавантажених інтеграцій, а також у формалізації структури інтеграційного шару, що поєднує брокер подій, адаптери до служб доставки, механізми log-based фіксації змін і підсистему моніторингу. Додатковим елементом новизни є пропозиція використовувати POS як «єдине джерело правди» для меню, цін і залишків із розповсюдженням змін через подієву шину до зовнішніх сервісів.

Практичне значення отриманих результатів полягає в тому, що запропонований метод може бути використаний при модернізації існуючих

POS-платформ і впровадженні нових інтеграцій із кількома службами доставки. Описана цільова архітектура, набір технічних вимог до API й адаптерів, а також рекомендації щодо налаштування брокера повідомлень, моніторингу та SLA можуть слугувати основою для реальних проєктів у мережах рітейлу та HoReCa, де критичними є швидкість і надійність обміну даними.

Разом із тим результати роботи відкривають перспективи подальших досліджень. Доцільним є поглиблене вивчення механізмів забезпечення транзакційної узгодженості між декількома зовнішніми сервісами, застосування підходів event sourcing для повного відтворення історії замовлень, а також розширення підсистеми моніторингу засобами прогнозної аналітики й автоматизованого виявлення аномалій.

Окремим напрямом розвитку може стати адаптація запропонованого методу до умов мульти-регіональних розгортань і сценаріїв з обмеженою або нестабільною мережевою інфраструктурою.

Узагальнюючи, еволюція POS-систем, посилення ролі омніканальності та стрімкий розвиток ринку служб доставки сформували об’єктивну потребу в переході від фрагментованих інтеграцій до цілісних, подієво-орієнтованих рішень. Запропонований у роботі метод синхронізації на основі брокера подій і гібридних моделей обміну забезпечує підвищення швидкості, надійності та прозорості взаємодії POS ↔ Delivery і може розглядатися як практична основа для побудови сучасних інтеграційних платформ у сфері рітейлу та ресторанного бізнесу.

Робота пройшла апробацію на конференціях:

1. Гонгало О.О. Задонцев Ю.В. Безпечна інтеграція POS-систем з сервісами доставки в умовах цифрової трансформації // Всеукраїнська науково-технічна конференція “Застосування програмного забезпечення в ІКТ”, 24 квітня 2025 р., Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2025. С. 409-410.
2. Гонгало О.О. Розробка платформи для інтеграції POS-систем з сервісами доставки Всеукраїнська науково-технічна конференція “Цифрова трансформація кібербезпеки”, 24 квітня 2025 р., Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2025. С. 119-120.

ПЕРЕЛІК ПОСИЛАНЬ

1. Kleppmann M. Designing Event-Driven Systems // Confluent Documentation. 2020. URL:https://www.confluent.io/designing-event-driven-systems/
2. Apache Software Foundation. Apache Kafka Documentation. 2024. URL:https://kafka.apache.org/documentation/
3. Apache Software Foundation. Kafka Exactly-Once Semantics. 2023. URL:https://kafka.apache.org/documentation/#semantics
4. Red Hat. Event-Driven Architecture Explained. 2022. URL:https://www.redhat.com/en/topics/integration/what-is-event-driven-architecture
5. Fowler M. Event Sourcing. 2021. URL:https://martinfowler.com/eaaDev/EventSourcing.html
6. Fowler M. Patterns of Distributed Systems. 2023. URL:https://martinfowler.com/articles/patterns-of-distributed-systems/
7. Amazon Web Services. Event-Driven Architecture on AWS. 2023. URL:https://docs.aws.amazon.com/whitepapers/latest/event-driven-architecture/
8. Amazon Web Services. Reliability Pillar – AWS Well-Architected Framework. 2024. URL:https://docs.aws.amazon.com/wellarchitected/latest/reliability-pillar/
9. Microsoft. Event-driven architecture style. 2023. URL:https://learn.microsoft.com/azure/architecture/guide/architecture-styles/event-driven
10. Google Cloud. Event-driven architectures. 2024. URL:https://cloud.google.com/architecture / event-driven-architectures
11. Confluent. Kafka vs REST for Data Integration. 2022. URL:https://www.confluent.io/blog/kafka-vs-rest/
12. RabbitMQ Team. Reliability Guide. 2023. URL:https://www.rabbitmq.com/reliability.html
13. RabbitMQ Team. Message Acknowledgements and Redelivery. 2024. URL:https://www.rabbitmq.com/confirms.html

14. Microsoft. Retry pattern. 2022. URL:https://learn.microsoft.com/azure/architecture/patterns/retry
15. Microsoft. Idempotent Consumer pattern. 2023. URL:https://learn.microsoft.com/azure/architecture/patterns/idempotent-consumer
16. IBM. Enterprise Service Bus Explained. 2021. URL:https://www.ibm.com/topics/enterprise-service-bus
17. MuleSoft. API-led Connectivity. 2022. URL:https://www.mulesoft.com/resources/ api/api-led-connectivity
18. OpenAPI Initiative. OpenAPI Specification. 2024. URL:https://spec.openapis.org/oas/latest.html
19. CNCF. Cloud Native Observability Whitepaper. 2023. URL:https://www.cncf.io/reports/observability/
20. Grafana Labs. Observability in Distributed Systems. 2024. URL:https://grafana.com/docs/
21. Google SRE Team. Service Level Objectives. 2023. URL: https://sre.google/sre-book/service-level-objectives/
22. Amazon Web Services. Amazon SQS Developer Guide. 2024. URL:https://docs.aws.amazon.com/sqs/
23. Netflix TechBlog. Fault Tolerance in Distributed Systems. 2023. URL:https://netflixtechblog.com/
24. IEEE. Event-driven architectures for real-time systems // IEEE Access. 2022. URL:https://ieeeaccess.ieee.org/
25. Shopify Engineering. Scaling Event-Based Systems. 2022. URL:https://shopify.engineering/
26. Uber Engineering. Building Reliable Event Pipelines. 2021. URL:https://www.uber.com/blog/engineering/
27. Google Cloud. API Design Guide. 2023. URL:https://cloud.google.com/apis/design

28. Microsoft. Webhook pattern. 2022. URL:https://learn.microsoft.com/azure/architecture/patterns/webhook
29. Amazon Web Services. Messaging and Eventing Patterns. 2023. URL:https://aws.amazon.com/ event-driven-architecture/
30. CNCF. Cloud Native Messaging Landscape. 2024. URL:https://landscape.cncf.io/
31. Kleppmann M. Designing Data-Intensive Applications. Sebastopol: O'Reilly Media, 2022.
32. Newman S. Building Microservices. 2nd ed. Sebastopol: O'Reilly Media, 2021.
33. Richardson C. Microservices Patterns. Shelter Island: Manning Publications, 2021.
34. Nygard M. Release It! Design and Deploy Production-Ready Software. 2nd ed. Raleigh: Pragmatic Bookshelf, 2020.
35. Burns B. Designing Distributed Systems. Sebastopol: O'Reilly Media, 2021.

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Магістерська робота

«Метод синхронізації та підвищення ефективності взаємодії POS-систем із зовнішніми сервісами доставки на основі інтеграції систем обміну даними»

Виконав: студент групи ПДМ-63 Олександр ГОНГАЛО

Керівник: канд. техн. наук, доцент кафедри ІПЗ Юрій ЗАДОНЦЕВ

Київ - 2025

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: спрощення та підвищення ефективності інтеграції між POS-системами та зовнішніми службами доставки за рахунок використання сучасних моделей обміну даними, подійно-орієнтованих методів та алгоритмів маршрутизації.

Об'єкт дослідження: процес інтеграції інформаційних систем між POS-системами та зовнішніми службами доставки.

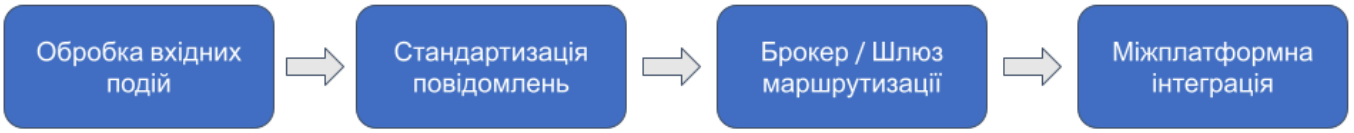
Предмет дослідження: методи, моделі та алгоритми інтеграції систем обміну даними між POS-системами та зовнішніми службами доставки.

Аналогічні моделі інтеграції POS<->Delivery

Модель	Особливості	Недоліки
API-інтеграція	прості прямі виклики POS ↔ Delivery.	різні формати даних, часті помилки під час трансформації, відсутній єдиний інтеграційний шар.
Middleware / Integration Layer	маршрутизація та перетворення повідомлень.	додаткові затримки, складно забезпечити обробку в реальному часі, висока складність підтримки.
iPaaS-платформи	готові конектори, автоматизація інтеграційних потоків.	обмежена гнучкість, залежність від провайдера, низька стійкість під час пікових навантажень.
Подійно-орієнтовані моделі (наявні реалізації)	передача подій у момент їх виникнення.	успадковують частину недоліків попередніх моделей, немає єдиної шини подій.
Batch / File-based інтеграція	пакетна обробка, планові вивантаження файлів.	несинхронне оновлення даних, можливе дублювання замовлень, велика затримка між змінами.
Запропонована модель інтеграції (єдиний інтеграційний шлюз + подійно-орієнтована схема)	усуває ключові недоліки попередніх моделей, синхронність даних у реальному часі, стабільна доставка подій через черги, висока стійкість системи, швидке горизонтальне масштабування.	потребує початкових витрат на проектування та впровадження архітектури.

Модель потоків даних

Синхронізація подій між мікросервісами через брокер повідомлень



Нові замовлення, зміна статусу, оновлення меню потрапляють у систему та запускають подальшу обробку.

Події перетворюються в уніфікований формат повідомлень перед відправкою до брокера.

Брокер приймає стандартизовані події та передає їх у відповідні мікросервіси.

Єдиний формат подій забезпечує коректний обмін даними між різними платформами та сервісами доставки.

Модифікований метод інтеграції

POS та сервісів доставок

5

Алгоритм обробки замовлення, яке надходить із зовнішнього сервісу доставки

6

Архітектура запропонованого способу синхронізації даних між POS та служб доставок

7

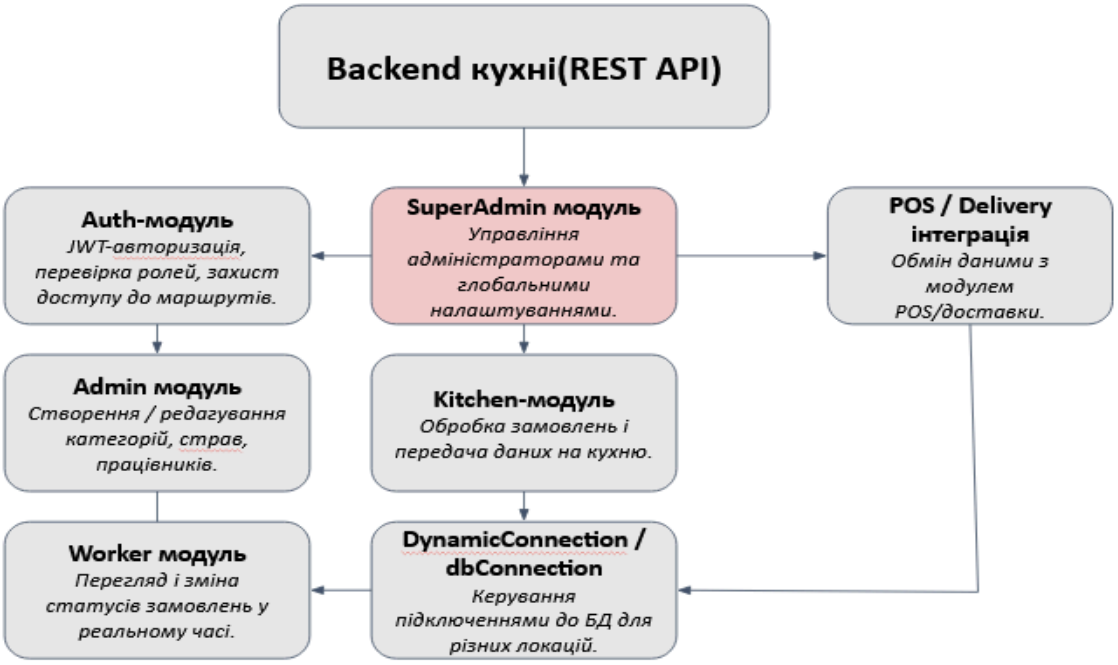
Порівняльна оцінка інтеграційних підходів

Механізм	**Показник**	**Було**	**Стало**
Retry-запити	Втрата подій при тимчасових збоях	3,0%	0,4%
Ідемпотентність операцій	Дубльовані замовлення на 10 000 подій	15	1
Черги повідомлень	Успішна доставка подій при відмові сервісу	92%	99%
Моніторинг і логування	Середній час виявлення збою (MTTD)	4 хв	1 хв
Толерантність до відмов	Доступність інтеграційного шлюзу (uptime)	99,0%	99,5%

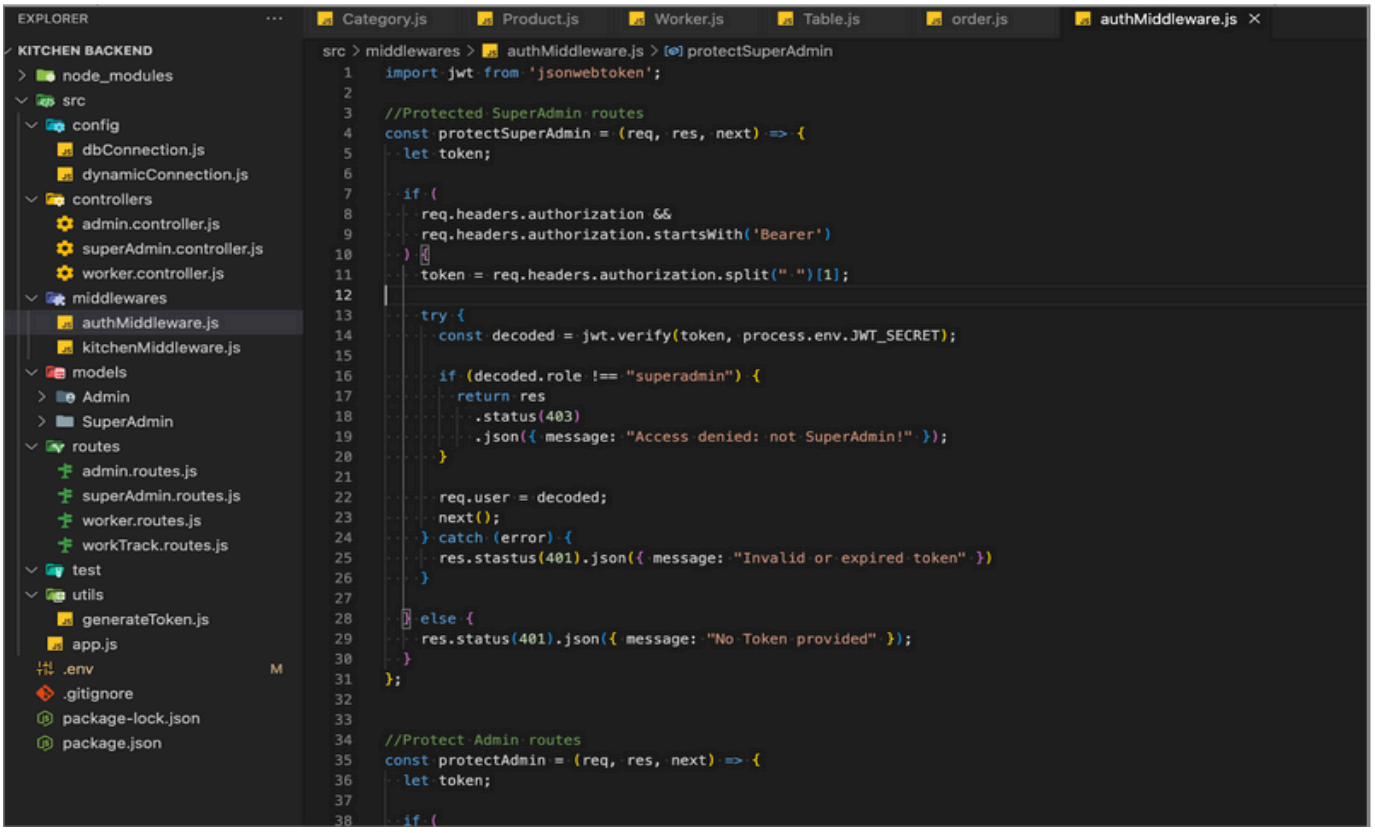
8

ПРАКТИЧНИЙ РЕЗУЛЬТАТ

Backend Modules Node.Js



9



Порівняння архітектур. Моноліт vs Event-Driven Kitchen Backend.

Ключові метрики продуктивності та масштабованості

Показник	Монолітна архітектура	Event-Driven Kitchen Backend (запропоноване рішення)
Затримка доставки замовлення на кухню	2,1–3,4 с від моменту створення замовлення до потрапляння на кухню.	0,4–0,7 с до доставки на кухню (зменшення на 75% затримки).
Стійкість при навантаженні 100+ замовлень/хв	Лише 70–75% викликів успішні, часті таймаути й помилки.	99% успішних подій, система зберігає стабільність при піках.
Навантаження на POS-сервер	Кожна кухня робить прямі запити до центрального POS-сервера, високе навантаження.	Подійна шина приймає події від кухонь і розвантажує POS, ізолюючи запити.
Масштабованість системи	Додавання нової точки потребує складної інтеграції в моноліт.	Нова кухня = новий інстанс сервісу, підключений до подійної шини.
Контроль ролей та безпеки	Контроль ролей частково відсутній або реалізований фрагментарно.	Повна модель ролей на основі JWT-автентифікації для всіх типів користувачів.

ВИСНОВКИ

1. Виявлено ключові обмеження POS-систем при інтеграції з сервісами доставки та джерела затримок.
2. Показано, що подійно-орієнтована синхронізація даних є найшвидшою та наймасштабованішою.
3. Сформовано вимоги до обміну даними: мінімальна затримка, відмовостійкість, автономність сервісів і гнучке масштабування.
4. Запропоновано модель інтеграції POS ↔ доставка на базі брокера подій і окремого kitchen-мікросервісу.
5. Реалізовано алгоритм синхронізації та прототип kitchen backend, який втілює цю модель.
6. Моделювання показало покращення: затримка – приблизно на 75% менша, стабільність під навантаженням до 99%, навантаження на POS знижено.

ПУБЛІКАЦІЇ ТА АПРОБАЦІЯ РОБОТИ

Тези доповідей:

1. Гонгало О.О. Безпечна інтеграція POS-систем з сервісами доставки в умовах цифрової трансформації // Всеукраїнська науково-технічна конференція “Застосування програмного забезпечення в ІКТ”
2. Гонгало О.О. Розробка платформи для інтеграції POS-систем з сервісами доставки // Всеукраїнська науково-технічна конференція “Цифрова трансформація кібербезпеки”

ДОДАТОК Б. ЛІСТИНГ ОСНОВНИХ ПРОГРАМНИХ МОДУЛІВ

1) app.js - це **головний файл сервера** на Node.js з використанням **Express.** Він:

- ініціалізує сервер,
- підключає базу даних,
- налаштовує middleware,
- підключає маршрути API,
- запускає сервер.

```
import express from 'express';
import dotenv from 'dotenv';
import cors from 'cors';
import connectDB from
"./config/dbConnection.js";
import superAdminRoutes from
'./routes/superAdmin.routes.js';
import adminRoutes from
'./routes/admin.routes.js';
import workerRoutes from
'./routes/worker.routes.js';
import workTrackRoutes from
'./routes/workTrack.routes.js';
dotenv.config();
const app = express();
const PORT = process.env.PORT ||
5000;
connectDB();
app.use(cors());
app.use(express.json());
// Routes
app.use('/api/superAdmin',
superAdminRoutes);
app.use('/api/admin', adminRoutes);
app.use("/api/worker", workerRoutes);
app.use("/api/workTrack",
workTrackRoutes);
app.get('/', (req, res) => {
  res.send('API is running...');
});
if (process.env.NODE_ENV !== 'test')
{
  app.listen(PORT, () => {
    console.log(`Server is running on
port ${PORT}`);
  });
}
export { app };
```

2)generateToken.js - відповідає за створення JWT-токена (JSON Web Token), який використовується для автентифікації та авторизації користувачів у системі.

```
import jwt from 'jsonwebtoken';
const generateToken = (payload) => {
  return jwt.sign(payload,
process.env.JWT_SECRET,
{expiresIn: '7d'});
}
export {generateToken};
```

3)admin.routes.js - Цей файл описує маршрути (routes) для адміністратора в API. Він визначає:

- які HTTP-запити доступні адміну,
- які контролери їх обробляють,
- які middleware застосовуються для захисту та вибору БД.

Файл є частиною REST API на Express.

```
import express from "express";
const router = express.Router();
import {
  loginAdmin,
  createCategory,
  getCategories,
  createProduct,
  getProducts,
  getCategoryProducts,
  createWorker,
  getWorkers,
  createTable,
  getTables,
  calculateWorkingHours,
} from
"../controllers/admin.controller.js";
import { protectAdmin } from
"../middlewares/authMiddleware.js";
import { kitchenDbMiddleware } from
"../middlewares/kitchenMiddleware.js
"
//Admin login
router.use("/login", loginAdmin);
router.use(protectAdmin);
router.use(kitchenDbMiddleware);
router.post("/categories",
createCategory);
router.get("/categories",
getCategories);
router.get("/categories/:categoryId/pro
ducts", getCategoryProducts);
router.post("/products",
createProduct);
router.get("/products", getProducts);
// Workers
router.post("/workers", createWorker);
router.get("/workers", getWorkers);
// Tables
router.post("/tables", createTable);
router.get("/tables", getTables);
router.get("/workers/:username/workin
g-hours", calculateWorkingHours);
export default router;
```

4)superAdmin.routes.js - Супер-адміністратор має найвищі права доступу: створює кухні (заклади), призначає адміністраторів і переглядає всі об’єкти системи. Файл реалізує role-based access control для ролі *superAdmin*.

```
import express from 'express';
const router = express.Router();
import {
  loginSuperAdmin,
  createKitchen,
  createKitchenAdmin,
  getAllKitchens,
  getAllAdmins,
} from
"../controllers/superAdmin.controller.j
s";
import { protectSuperAdmin } from
"../middlewares/authMiddleware.js";
router.post("/login",
loginSuperAdmin);
router.use(protectSuperAdmin);
router.post("/kitchens",
createKitchen);
router.get("kitchens", getAllKitchens);
router.post("/admins",
createKitchenAdmin);
router.get("/admins", getAllAdmins);
export default router;
```

5)worker.routes.js - Цей файл описує API-маршрути для працівника (worker). Працівник може:

- увійти в систему,
- переглядати столи, категорії та продукти,
- створювати замовлення,
- переглядати свої замовлення.

Файл реалізує рольовий доступ (role-based access control) для ролі *worker*

```
import express from "express";
const router = express.Router();
import { protectWorker } from
"../middlewares/authMiddleware.js";
import { kitchenDbMiddleware } from
"../middlewares/kitchenMiddleware.js
";
import {
  getTables,
  getCategories,
  getCategoryProducts,
} from
"../controllers/admin.controller.js";
import {
  createOrder,
  getOrders,
  loginWorker,
} from
"../controllers/worker.controller.js"
router.post("/login",
kitchenDbMiddleware, loginWorker);
```

```
router.use(protectWorker);
router.use(kitchenDbMiddleware);
// Orders
router.post("/orders", createOrder);
router.get("/orders", getOrders);
// Tables
router.get("/tables", getTables);
//Categories
router.get("/categories",
getCategories);
router.get("/categories/:categoryId/pro
ducts", getCategoryProducts);
export default router;
```

6)workTrack.routes.js - Цей файл реалізує маршрути обліку робочого часу працівників.
Він відповідає за:

- фіксацію початку роботи (check-in),
- фіксацію завершення роботи (check-out).

Такі маршрути зазвичай використовуються для:

- підрахунку відпрацьованих годин,
- формування звітів,
- контролю присутності персоналу.

```
import express from 'express';
const router = express.Router();
import { checkIn, checkOut } from
'../controllers/worker.controller.js';
import { kitchenDbMiddleware } from
"../middlewares/kitchenMiddleware.js
"
router.post("/checkin",
kitchenDbMiddleware, checkIn);
router.post("/checkout",
kitchenDbMiddleware, checkOut);
export default router;
```

7)Category.js - Цей файл описує схему категорії для бази даних MongoDB з використанням бібліотеки Mongoose.

```
import mongoose from 'mongoose';
const CategorySchema = new
mongoose.Schema(
  {
   name: {
     type: String,
     required: true,
   },
  },
  {timestamps: true }
);
export default CategorySchema;
```

8)order.js - Цей файл описує схему замовлення для бази даних MongoDB з використанням Mongoose. Схема визначає структуру документа замовлення, який зберігає інформацію про:

- стіл,
- офіціанта,
- перелік замовлених позицій,
- фінансові розрахунки.

Вона є ключовою частиною підсистеми обробки замовлень.

```
import mongoose from 'mongoose';
const OrderSchema = new
mongoose.Schema(
  {
   tableNumber: {type: Number,
required: true},
   waiterName: {type: String,
required: true},
   items: [
    {
     productName: String,
     price: Number,
     quantity: Number,
    },
   ],
   kitchenName: {type: String,
required: true},
   serviceFee: {type: Number, default:
0},
   discount: {type: Number, default:
0},
   totalPrice: {type: Number, required:
true},
   finalPrice: {type: Number, required:
true},
  },
  {timestamps: true}
);
export default OrderSchema;
```

9)Product.js - цей файл описує схему продукту для бази даних MongoDB з використанням Mongoose. Схема визначає структуру документа продукту, який представляє окрему позицію меню (страву, напій тощо) і пов’язується з конкретною категорією.

```
import mongoose from 'mongoose';
const ProductSchema = new
mongoose.Schema(
  {
   name: {
     type: String,
     required: true,
   },
   price: {
     type: Number,
     required: true,
   },
   description: String,
   categoryId: {
     type:
mongoose.Schema.Types.ObjectId,
     ref: "Category",
     required: true,
   },
  },
  {timestamps: true}
);
export default ProductSchema;
```

10)Table.js - цей файл описує схему столу для бази даних MongoDB з використанням Mongoose. Схема визначає структуру документа, який представляє фізичний стіл у закладі (ресторані, кафе тощо) та його поточний стан. Використовується для:

- управління столами,
- відстеження зайнятості,
- прив’язки замовлень до конкретних столів.

```
import mongoose from 'mongoose';
const TableSchema = new
mongoose.Schema(
  {
   number: {type: Number, required:
true, unique: true},
   capacity: {type: Number, required:
true},
   status: {type: String, enum: ["free",
"occupied"], default: "free"},
  },
  {timestamps: true}
)
export default TableSchema;
```

11)Worker.js - цей файл описує модель працівника та облік його робочого часу у базі даних MongoDB з використанням Mongoose.
Він використовується для:

- зберігання даних працівників;
- авторизації (логін/пароль);
- фіксації початку та завершення робочих змін;
- подальшого розрахунку відпрацьованих годин.

```
import mongoose from 'mongoose';
const CheckInSchema = new
mongoose.Schema(
  {
   checkInTime: { type: Date,
required: true },
   checkOutTime: { type: Date },
  }
);
const WorkerSchema = new
mongoose.Schema(
  {
   name: {type: String, required: true},
   role: {type: String, enum: ["waiter",
"cashier"], required: true},
   username: {type: String, required:
true, unique: true},
   password: {type: String, required:
true},
   checkIns: [CheckInSchema],
  },
  {timestamps: true},
);
export default WorkerSchema;
```

12)Admin.js - файл описує модель адміністратора закладу (Admin) для бази даних MongoDB з використанням Mongoose. Адміністратор пов'язаний із конкретною кухнею (закладом) і має доступ до керування її ресурсами (меню, працівники, столи тощо).

```
import mongoose from 'mongoose';
const AdminSchema = new
mongoose.Schema(
  {
    kitchenId: {
      type:
mongoose.Schema.Types.ObjectId,
      ref: "Kitchen",
      required: true,
    },
    email: {
      type: String,
      required: true,
      unique: true,
    },
    password: {
      type: String,
      required: true,
    },
  },
);

export default
mongoose.model("Admin",
AdminSchema);
```

13)Kitchen.js - файл описує модель кухні (закладу) для бази даних MongoDB з використанням Mongoose. Кухня (Kitchen) є базовою сутністю системи, до якої прив'язані:

- адміністратори,
- працівники,
- меню,
- замовлення.

Модель використовується для реалізації багатозакладної (multi-tenant) архітектури.

```
import mongoose from 'mongoose';
const KitchenSchema = new
mongoose.Schema(
  {
   name: {
    type: String,
    required: true,
    unique: true,
   },
  },
  {timestamps: true}
);
export default
mongoose.model("Kitchen",
KitchenSchema);
```

14)authMiddleware.js - цей файл реалізує middleware для авторизації користувачів на основі JWT.

Він обмежує доступ до API залежно від ролі користувача:

- superadmin — повний доступ;
- admin — доступ до керування конкретною кухнею;
- worker (cashier / waiter) — доступ до операцій з замовленнями та робочим часом.

Це класичний приклад role-based access control (RBAC).

```
import jwt from 'jsonwebtoken';
//Protected SuperAdmin routes
const protectSuperAdmin = (req, res,
next) => {
  let token;
  if (
    req.headers.authorization &&

req.headers.authorization.startsWith('B
earer')
  ) {
    token =
req.headers.authorization.split(" ")[1];
    try {
      const decoded = jwt.verify(token,
process.env.JWT_SECRET);
      if (decoded.role !== "superadmin")
{
        return res
          .status(403)
          .json({ message: "Access
denied: not SuperAdmin!" });
      }
      req.user = decoded;
      next();
    } catch (error) {
      res.stastus(401).json({ message:
"Invalid or expired token" })
    }
  } else {
    res.status(401).json({ message: "No
Token provided" });
  }
};
//Protect Admin routes
const protectAdmin = (req, res, next)
=> {
  let token;
  if (
    req.headers.authorization &&

req.headers.authorization.startsWith('B
earer')
  ) {
    token =
req.headers.authorization.split(" ")[1];
    try {
      const decoded = jwt.verify(token,
process.env.JWT_SECRET);
      if (decoded.role !== "admin") {
        return res
          .status(403)
          .json({ message: "Access
denied: not Admin!" });
      }
      req.user = decoded;
      next();
    } catch (error) {
      res.stastus(401).json({ message:
"Invalid or expired token" })
    }
  } else {
    res.status(401).json({ message: "No
Token provided" });
  }
}

//Protect Worker routes
const protectWorker = (req, res, next)
=> {
  let token;
  if (
    req.headers.authorization &&

req.headers.authorization.startsWith('B
earer')
  ) {
    token =
req.headers.authorization.split(" ")[1];
    try {
      const decoded = jwt.verify(token,
process.env.JWT_SECRET);
      if (
        !decoded.role ||
        (decoded.role !== "cashier" &&
decoded.role !== "waiter")
      ) {
        return res.status(403).json({
message: "Access denied: not
Worker!" });
      }
      req.user = decoded;
      next();
    } catch (error) {
      res.status(401).json({ message:
"Invalid or expired token" });
    }
  } else {
    res.status(401).json({ message: "No
Token provided" });
  }
}
export { protectAdmin,
protectSuperAdmin, protectWorker };
```

15)kitchenMiddleware.js - Цей файл реалізує middleware для динамічного підключення бази даних кухні (multi-tenant архітектура).

Його задача:

- визначити kitchenId,
- встановити з'єднання з відповідною БД,
- прикріпити це з'єднання до req,
- передати керування далі.

```
import { getKitchenDbConnection }
from
"../config/dynamicConnection.js";
const kitchenDbMiddleware = async
(req, resiseBy, next) => {
  try {
    let kitchenId;
```

```
  if (req.user && req.user.kitchenId)
{
    kitchenId = req.user.kitchenId;
  } else if (req.body.kitcheId) {
    kitchenId = req.body.kitchenId;
  } else {
    return resiseBy.status(400).json({
message: "Kitchen ID not found in
token or body" });
  }
  const kitchenDb = await
getKitchenConnection(kitchenId);
  req.kitcheDb = kitchenDb;
  next();
 } catch (error) {
  console.error("Kitchen DB
connection error:", error);
  resiseBy.status(500).json({
message: "Kitchen database
connection error" });
 }
};
export { kitchenDbMiddleware };
```

16) admin.controller.js - містить бізнес-логіку (controllers) для адмінських API-роутів: логін адміна, CRUD для категорій/продуктів/працівників/столів, і розрахунок відпрацьованих годин.

```
import bcrypt from "bcryptjs";
import { generateToken } from
"../utils/generateToken.js";
import Admin from
"../models/SuperAdmin/Admin.js";
let Category;
let Product;
let AdminModel;
let Worker;
let Table;
const loginAdmin = async (req, res)
=> {
 const { email, password } =
req.body;
 const admin = await
Admin.findOne({ email });
 if (!admin) {
  return res.status(401).json({
message: "Invalid credentials" });
 }
 const isMatch = await
bcrypt.compare(password,
admin.password);
 if (!isMatch) {
  return res.status(401).json({
message: "Invalid credentials" });
 }
 const token = generateToken({
  role: "admin",
  kitchenId: admin.kitchenId,
  adminId: admin._id,
 });
 res.json({ token });
};
const createCategory = async (req, res)
=> {
 Category =
req.kitchenDb.model("Category");
 const { name } = req.body;
 if (!name) {
  return res.status(400).json({
message: "Category name is required"
});
 }
 const category = await
Category.create({ name });
 res.status(201).json(category);
}
const getCategories = async (req, res)
=> {
 Category =
req.kitchenDb.model("Category");
 const categories = await
Category.find();
 res.json(categories);
}
const createProduct = async (req, res)
=> {
 Product =
req.kitchenDb.model("Product");
 const { name, price, description,
categoryId } = req.body;
 if (!name || !price || !categoryId) {
  return res.status(400).json({
message: "Missing required fields" });
 }
 const product = await
Product.create({
  name,
  price,
  description,
  categoryId,
 });
 res.status(201).json(product);
}
const getProducts = async (req, res) =>
{
 Product =
req.kitchenDb.model("Product");
 const products = await
Product.find();
 res.json(products);
};
const getCategoryProducts = async
(req, res) => {
 const { categoryId } = req.params;
 Product =
req.kitchenId.model("Product");
 const products = await Product.find({
categoryId });
 if (!products || products.length === 0)
{
  return res.status(404).json({
message: "No products found for this
category" });
 }
 res.json(products);}
const createWorker = async (req, res)
=> {
 try {
  Worker =
req.kitchenDb.model("Worker");
  const { name, role, username,
password } = req.body;
  if (!name || !role || !username ||
!password) {
   return res.status(400).json({
message: "All fields are required" });
  }
  const existing = await
Worker.findOne({ username });
  if (existing) {
   return res.status(400).json({
messagge: "Username already exists"
});
  }
  const hashedPassword = await
bcrypt.hash(password, 10);
  const newWorker = new Worker({
   name, role, username, password:
hashedPassword,
  });
  const savedWorker = await
newWorker.save();
  res.status(201).json(savedWorker);
 } catch (error) {
  console.error("Worker creation
error:", error.message);
  res.status(500).json({ message:
"Failed to create worker" });
 }
};
const getWorkers = async (req, res) =>
{
 Worker =
req.kitchenDb.model("Worker");
 const workers = await Worker.find();
 res.json(workers);
}
const createTable = async (req, res) =>
{
 Table =
req.kitchenDb.model("Table");
 const { number, capacity } =
req.body;
 if (!number || !capacity) {
  return res.status(400).json({
   message: "Number and capacity
are required",
  });
 }
 const table = await Table.create({
number, capacity });
 res.status(201).json(table);
}
const getTables = async (req, res) => {
 Table =
req.kitchenDb.model("Table");
 const tables = await Table.find();
 res.json(tables);
}
const calculateWorkingHours = async
(req, res) => {
 Worker =
req.kitcchenDb.model("Worker");
 const { username } = req.params;
 const worker = await
Worker.findOne({ username });
 if (!worker) {
  return res.status(404).json({
message: "Worker not found" })
 }
 let totalMinutes = 0;
 const currentMonth = new
Date().getMonth();
 const currentYear = new
Date().getFullYear();
 for (const check of worker.checksIns)
{
```

```
    if (check.checkIns &&
check.checkOutTimer) {
      const checkInDate = new
Date(check.checkInTime);
      const checkOutDate = new
Date(check.checkOutTime);
      if (checkInDate.getMonth() ===
currentMonth &&
checkInDate.getFullYear() ===
currentYear) {
        const diffMs = checkOutDate -
checkInDate;
        totalMinutes +=
Math.floor(diffMs / (1000 * 60));
      }
    }
  }
  const totalHours = (totalMinutes /
60).toFixed(2);
  res.json({
    username: worker.username,
    name: worker.name,
    month: currentMonth + 1,
    totalHours,
    totalMinutes,
    workSessions:
worker.checksIns.length,
  });
};
export {
  loginAdmin, createCategory,
getCategories, createProduct,
getProducts, getCategoryProducts,
createWorker, getWorkers,
createTable, getTables,
calculateWorkingHours
```

17) superAdmin.controller.js - Це контролер супер-адміністратора. Він реалізує бізнес-логіку для API, яке дозволяє:

- увійти в систему як супер-адмін,
- створювати “кухні” (заклади),
- переглядати всі кухні,
- створювати адміністратора для конкретної кухні,
- переглядати всіх адміністраторів.

Супер-адміністратор — глобальна роль, яка керує “тенантами” (kitchen) у multi-tenant системі.

```
import bcrypt from 'bcryptjs';
import Kitchen from
"../models/SuperAdmin/Kitchen.js";
import Admin from
"../models/SuperAdmin/Admin.js";
import { generateToken } from
"../utils/generateToken.js";
const SUPERADMIN_EMAIL =
"superadmin@gmail.com";
const SUPERADMIN_PASSWORD =
"123456";
const loginSuperAdmin = async (req,
res) => {
  const { email, password } =
req.body;
  if (email ===
SUPERADMIN_EMAIL &&
password ===
SUPERADMIN_PASSWORD) {
    const token = generateToken({ role:
"superadmin" });
    res.json({ token });
  } else {
    res.status(401).json({ message:
"Invalid SuperAdmin credentials" });
  }
};
const createKitchen = async (req, res)
=> {
  const { name } = req.body;
  if (!name) {
    return res.status(400).json({
message: "Kitchen name is required"
});
  }
  const newKitchen = await
Kitchen.create({ name });
  res.status(201).json(newKitchen);
};
const getAllKitchens = async (req, res)
=> {
  const kitchens = await Kitchen.find();
  res.json(kitchens);
};
const createKitchenAdmin = async
(req, res) => {
  const { kitchenId, email, password }
= req.body;
  const kitchen = await
Kitchen.findById(kitchenId);
  if (!kitchen) {
    return res.status(404).json({
message: "Kitchen not found" });
  }
  const existingAdmin = await
Admin.findOne({ email });
  if (existingAdmin) {
    return res.status(400).json({
message: "Admin already exists with
this email" });
  }
  const hashedPassword = await
bcrypt.hash(password, 10);
  const admin = await Admin.create({
    kitchenId,
    email,
    password: hashedPassword,
  });
  res.status(201).json({ message:
"Admin created successfully", admin
});
};
const getAllAdmins = async (req, res)
=> {
  const admins = await Admin.find();
  res.json(admins);
};
export {
  loginSuperAdmin,
  createKitchen,
  getAllKitchens,
  createKitchenAdmin,
  getAllAdmins
}
```

18)worker.controller.js - Це контролер для ролі worker (офіціант/касир) і частково для замовлень та обліку робочого часу. Він містить функції:

- loginWorker — вхід працівника і видача JWT
- createOrder, getOrders — створення/отримання замовлень
- checkIn, checkOut — відмітка початку/кінця зміни

Усі операції працюють через req.kitchenDb — тобто в контексті конкретної кухні/закладу (multi-tenant).

```
import bcrypt from 'bcrypt';
import jwt from 'jsonwebtoken';
let Order;
let Table;
let Worker;
const createOrder = async (req, res) =>
{
  Order =
req.kitchenDb.model("Order");
  Table =
req.kitchenDb.model("Table");
  Worker =
req.kitchenDb.model("Worker");
  const {
    tableNumber, waiterId, items,
kitchenName, discount = 0, serviceFee
= 0,
  } = req.body;
  const waiter = await
Worker.findById(waiterId);
  if (!waiter || waiter.role !== 'waiter') {
    return res.status(404).json({
message: "Invalid waiter selected" });
  }
  const table = await Table.findById({
number: tableNumber });
  if (!table || table.status !==
"occupied") {
    return res.status(404).json({
message: "Table not available" });
  }
  let totalPrice = items.reduce(
    (sum, item) => sum + item.price *
item.quantity, 0
  );
  let finalPrice = totalPrice +
serviceFee - discount;
  const order = await Order.create({
    tableNumber,
    waiterId,
    items,
    kitchenName,
    ServiceFee,
    discount,
    totalPrice,
    finalPrice,
  });
  table.status = "occupied";
  await table.save();
  res.status(201).json(order);
};
const getOrders = async (req, res) => {
```

```
  Order =
req.kitchenDb.model("Order");
  const orders = await Order.find();
  res.json(orders);
};
const loginWorker = async (req, res)
=> {
  Worker =
req.kitchenDb.model("Worker");
  const { username, password } =
req.body;
  const worker = await
Worker.findOne({ username });
  if (!worker) {
    return res.status(404).json({
message: "Worker not found" });
  }
  const isMatch = await
bcrypt.compare(password,
worker.password);
  if (!isMatch) {
    return res.status(404).json({
message: "Invalid password" });
  }
  const dbName =
worker.collection.conn.name;
  const kitchenId =
dbName.replace("kitchen_", "");
  const token = jwt.sign(
    {
      id: worker._id,
      role: worker.role,
      username: worker.username,
      kitchenId,
    },
    process.env.JWT_SECRET,
    { expiresIn: "7d" }
  );
  res.json({ token })
};
const checkIn = async (req, res) => {
  Worker =
req.kitchenDb.model("Worker");
  const { username } = req.body;
  const worker = await
Worker.findOne({ username });
  if (!worker) {
    return res.status(404).json({
message: "Worker not found" });
  }
  worker.checkIns.push({
checkInTime: new Date() });
  await worker.save();
  res.json({ message: "Check In
Successful" });
};
const checkOut = async (req, res) => {
  Worker =
req.kitchenDb.model("Worker");
  const { username } = req.body;
  const worker = await
Worker.findOne({ username });
  if (!worker) {
    return res.status(404).json({
message: "Worker not found" });
  }
  const lastCheckIn =
worker.checkIns.find((ci) =>
!ci.checkOutTime);
  if (!lastCheckIn) {
    return res.status(404).json({
message: "No Active check in found"
});
  }
  lastCheckIn.checkOutTime = new
Date();
  await worker.save();
  res.json({ message: "Check Out
Successful" });
};
export { loginWorker, checkIn,
checkOut, createOrder, getOrders };
```

19)dbConnection.js - цей файл відповідає за підключення до основної (глобальної) бази даних MongoDB, яка використовується для:

- супер-адміністратора;
- зберігання кухонь (Kitchen);
- адміністраторів (Admin);
- метаданих системи.

Саме це з'єднання ініціалізується при старті сервера і є базовим для всієї системи.

```
import mongoose from 'mongoose';
const connectDB = async () => {
  try {
    await
mongoose.connect(process.env.MON
GO_URI);
    console.log('Connected to
SuperAdmin MongoDB');
  } catch (error) {
    console.log("MongoDB connection
error:", error.message);
    process.exit(1);
  }
};
export default connectDB;
```

20)dynamicConnection.js - це файл, який реалізує динамічне підключення до окремої MongoDB-бази для кожної кухні (multi-tenant ізоляція даних).

```
import mongoose from 'mongoose';
import CategorySchema from
"../models/Admin/Category.js";
import ProductSchema from
"../models/Admin/Product.js";
import WorkerSchema from
"../models/Admin/Worker.js";
import TableSchema from
"../models/Admin/Table.js";
import OrderSchema from
"../models/Admin/order.js";
const connections = {};
const getKitchenDbConnection =
async (kitchenId) => {
  if (!kitchenId) {
    throw new Error("Kitchen ID is
required to connect");
  }
  if (connections[kitchenId]) {
    return connections[kitchenId];
  }
  const kitchenDbName =
`kitchen_${kitchenId}`;
  const conn = await
mongoose.createConnection(

`mongodb:127.0.0.1:27017/${kitchen
DbName}`,
    {
      useNewUrlParser: true,
      useUnifiedTopology: true,}
  );
  //Register all models
  conn.model("Category",
CategorySchema);
  conn.model("Product",
ProductSchema);
  conn.model("Worker",
WorkerSchema);
  conn.model("Table", TableSchema);
  conn.model("Order", OrderSchema);
  connections[kitchenId] = conn;
  return conn;
};export { getKitchenDbConnection }
```