МАГІСТЕРСЬКА РОБОТА

МР.ІПм - 48.00.00.000- ПЗ

Група ІПм-24-2

Мердак Віталій

2025

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Мердак Віталій Богданович

(прізвище, ім'я, по батькові)

УДК 004.8

(індекс)

МАГІСТЕРСЬКА РОБОТА

Моделі, методи та алгоритми автоматизації фінансових процесів на основі хмарних обчислень

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

Мердак В. Б.

(підпис, ініціали та прізвище здобувача освітнього ступеня)

Науковий керівник Козак Олексій Федорович, канд. тех. н, ст. викл.

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Допущено до захисту

Завідувач кафедри

доц.			Бандура В. В.
(посада)	(підпис)	(дата)	(ініціали та прізвище)

Нормоконтроль

доц.			Вовк Р. Б.
(посада)	(підпис)	(дата)	(ініціали та прізвище)

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

Івано-Франківськ – 2025

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Освітньо-кваліфікаційний рівень магістр

Спеціальність 121 – інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Завідувач кафедри ІПЗ
доц. Бандура В. В.
„___” _____ 2025 р.

ЗАВДАННЯ

НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

Мердаку Віталію Богдановичу

(прізвище, ім’я, по батькові)

1. Тема магістерської роботи „Моделі, методи та алгоритми автоматизації фінансових процесів на основі хмарних обчислень”

керівник проєкту (роботи) Козак Олексій Федорович, к.т.н., старший викладач

затверджені наказом вищого навчального закладу від „05” 11 2025 р. № 695/7

2. Строк здачі студентом закінченої роботи 15 грудня 2025 р.

3. Вихідні дані до роботи Методи об’єктно-орієнтованого аналізу й проєктування, принципи побудови розподілених та хмарних систем, архітектурні патерни (Clean Architecture, Event-Driven Architecture), документація AWS, специфікації REST API та вимоги до автоматизації фінансового документообігу

4. Зміст пояснювальної записки (перелік питань, що їх належить розробити)

1. Аналіз проблеми автоматизації фінансових процесів та особливостей використання безсерверних технологій
2. Формальне моделювання та проєктування архітектури системи на основі хмарних сервісів
3. Реалізація та валідація прототипу системи автоматизації фінансових розрахунків

5. Перелік графічного матеріалу (з точним забезпеченням обов’язкових креслень)

1. Узагальнена структурна схема та діаграма потоків даних (Data Flow) системи
2. Блок-схема алгоритму розрахунку
3. Діаграма послідовності взаємодії компонентів
4. Динаміка обробки запитів під навантаженням

6. Консультанти з роботи із зазначенням розділів роботи, що їх стосуються

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Перевірка на плагіат	доц. к.т.н. Вовк Р. Б.		

7. Дата видачі завдання 3 листопада 2025 р.

Керівник _____ _____
(підпис) (розшифровка підпису)

Завдання прийняв до виконання _____ _____
(підпис) (розшифровка підпису)

КАЛЕНДАРНИЙ ПЛАН

Номер і назва етапів магістерської роботи	Термін виконання етапів роботи	Примітка
1 Підбір і вивчення літератури	03.11.2025 – 05.11.2025	виконано
2 Дослідження предметної області та постановка задачі	06.11.2025 – 16.11.2025	виконано
3 Концептуальне проєктування інструменту візуального модельно-орієнтованого інжинірингу	17.11.2025 – 24.11.2025	виконано
4 Реалізація технічної архітектури та механізмів інструменту	25.11.2025 – 30.11.2025	виконано
5 Експериментальна оцінка та підготовка прикладів використання інструменту	01.12.2025 – 07.12.2025	виконано
6 Оформлення пояснювальної записки згідно з вимогами	06.11.2025 – 14.12.2025	виконано
7 Затвердження пояснювальної записки роботи завідувачем кафедри	15.12.2025	виконано

Студент – магістр _____
(підпис) (розшифровка підпису)

Керівник проєкту _____
(підпис) (розшифровка підпису)

АНОТАЦІЯ

Магістерська робота: 91 с., 23 рис., 3 табл., 40 джерел.

Тема: Моделі, методи та алгоритми автоматизації фінансових процесів на основі хмарних обчислень.

Об'єкт дослідження: процеси автоматизації фінансового обліку та генерації платіжних документів у корпоративних інформаційних системах.

Мета роботи: підвищення ефективності та надійності процесу білінгу шляхом розробки автоматизованої системи на основі безсерверної архітектури (Serverless), що забезпечує мінімізацію операційних витрат на інфраструктуру та автоматичне масштабування під час пікових навантажень.

Предмет дослідження: методи та засоби побудови розподілених високонавантажених систем з використанням хмарних обчислень, черг повідомлень та подієво-орієнтованої архітектури.

Результати дослідження: Проведено аналіз ефективності традиційних та хмарних підходів до автоматизації фінансових процесів. Розроблено математичну модель обробки фінансових транзакцій та спроєктовано архітектуру системи на базі хмарної платформи AWS. Реалізовано програмний прототип системи з використанням платформи .NET 8, сервісів AWS Lambda, SQS та S3. Виконано експериментальну перевірку механізмів асинхронної генерації документів та їх доставки.

Висновок: В результаті досліджень було отримано робоче рішення, яке дозволяє усунути витрати на простій обчислювальних потужностей (idle resources), гарантує цілісність даних при асинхронній обробці та здатне автоматично адаптуватися до пікових навантажень, що підтверджує економічну та технічну доцільність переходу на безсерверні технології.

SERVERLESS, AWS LAMBDA, .NET, EVENT-DRIVEN ARCHITECTURE, БІЛІНГ, ХМАРНІ ОБЧИСЛЕННЯ, АСИНХРОННА ОБРОБКА, MICROSERVICES.

ANNOTATION

Master's Work: 91 p., 23 fig., 3 tab., 40 sources.

Topic: Development of an automated billing system using serverless architecture.

Object of research: processes of automation of financial accounting and generation of payment documents in corporate information systems.

Purpose: to increase the efficiency and reliability of the billing process by developing an automated system based on serverless architecture, ensuring minimization of operational infrastructure costs and automatic scaling during peak loads.

Subject of research: methods and tools for building distributed high-load systems using cloud computing, message queues, and event-driven architecture.

Research results: An analysis of the efficiency of traditional and cloud approaches to financial process automation was conducted. A mathematical model for processing financial transactions was developed, and a system architecture based on the AWS cloud platform was designed. A software prototype of the system was implemented using the .NET 8 platform, AWS Lambda, SQS, and S3 services. Experimental verification of asynchronous document generation and delivery mechanisms was performed.

Conclusion: As a result of the research, a working solution was obtained that eliminates the costs of idle computing resources, ensures data integrity during asynchronous processing, and is capable of automatically adapting to peak loads, confirming the economic and technical feasibility of transitioning to serverless technologies.

SERVERLESS, AWS LAMBDA, .NET, EVENT-DRIVEN ARCHITECTURE, BILLING, CLOUD COMPUTING, ASYNCHRONOUS PROCESSING, MICROSERVICES.

ЗМІСТ

РОЗДІЛ 3

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

API (Application Programming Interface) — прикладний програмний інтерфейс

AWS (Amazon Web Services) — хмарна платформа Amazon Web Services

CapEx (Capital Expenditures) — капітальні витрати

CI/CD (Continuous Integration / Continuous Deployment) — безперервна інтеграція та безперервне розгортання

CPU (Central Processing Unit) — центральний процесор

CRUD (Create, Read, Update, Delete) — функції створення, читання, оновлення та видалення даних

DTO (Data Transfer Object) — об'єкт передачі даних

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

FaaS (Function as a Service) — функція як послуга

HTTP (Hypertext Transfer Protocol) — протокол передачі гіпертексту

IaaS (Infrastructure as a Service) — інфраструктура як послуга

IAM (Identity and Access Management) — система управління ідентифікацією та доступом

JWT (JSON Web Token) — вебтокен JSON

ORM (Object-Relational Mapping) — об'єктно-реляційне відображення

PaaS (Platform as a Service) — платформа як послуга

PDF (Portable Document Format) — міжплатформний формат електронних документів

REST (Representational State Transfer) — передача репрезентативного стану

S3 (Simple Storage Service) — сервіс простого зберігання даних

SDK (Software Development Kit) — комплект засобів розробки програмного забезпечення

SLA (Service Level Agreement) — угода про рівень надання послуг

SMTP (Simple Mail Transfer Protocol) — простий протокол передачі електронної пошти

ВСТУП

Актуальність роботи

В умовах стрімкої цифровізації економіки та переходу підприємств на модель дистанційної роботи, критичного значення набувають питання автоматизації внутрішніх бізнес-процесів, зокрема фінансового обліку та білінгу. Традиційні підходи до обробки рахунків та нарахування заробітної плати, що базуються на ручному вводі даних або використанні застарілих монолітних систем, демонструють свою неефективність. Вони характеризуються високим ризиком людської помилки, значними часовими затримками та, що особливо важливо, нераціональним використанням обчислювальних ресурсів.

Особливістю процесів генерації інвойсів та фінансової звітності є їхня періодичність та нерівномірність навантаження. Як правило, ці операції виконуються один або кілька разів на місяць (наприкінці звітного періоду), створюючи короткочасні пікові навантаження на систему. Утримання виділеної серверної інфраструктури, яка працює в режимі 24/7, для обслуговування процесів, що активні лише кілька годин на місяць, є економічно невиправданим. У цьому контексті перехід до безсерверної архітектури (Serverless) та використання хмарних обчислень стає не просто технологічним трендом, а необхідною умовою для оптимізації операційних витрат (OpEx) [1].

Таким чином, розробка автоматизованої системи білінгу, побудованої на принципах подієво-орієнтованої архітектури (Event-Driven Architecture) з використанням хмарних сервісів, є актуальним науково-практичним завданням, що дозволяє поєднати надійність обробки даних з гнучким масштабуванням та мінімізацією витрат [2].

Мета і задачі дослідження

Метою роботи є підвищення ефективності та надійності процесу білінгу в організації шляхом розробки автоматизованої системи на основі безсерверної архітектури, що забезпечує мінімізацію витрат на інфраструктуру та автоматичне масштабування під час пікових навантажень.

Для досягнення поставленої мети необхідно вирішити такі завдання:

1) Провести аналіз існуючих підходів до автоматизації фінансових процесів та виявити недоліки традиційних архітектурних рішень.

2) Обґрунтувати доцільність використання безсерверних технологій (AWS Lambda, SQS) для задач з періодичним навантаженням.

3) Розробити математичну та структурну модель системи, включаючи схему потоків даних та взаємодію сутностей (співробітники, проєкти, інвойси).

4) Спроєктувати та реалізувати програмний прототип системи з використанням платформи .NET та хмарних сервісів AWS.

5) Виконати валідацію розробленого рішення, перевірити коректність генерації документів та надійність асинхронної доставки повідомлень.

Об’єктом дослідження є процеси фінансового обліку та генерації платіжних документів в корпоративних інформаційних системах.

Предметом дослідження є методи та засоби побудови масштабованих автоматизованих систем білінгу з використанням безсерверної архітектури та черг повідомлень.

Методи дослідження. У роботі використано методи системного аналізу для формування вимог до системи, методи об’єктно-орієнтованого проєктування та моделювання баз даних для побудови архітектури, а також методи експериментального дослідження для тестування продуктивності та надійності програмного забезпечення.

Наукова новизна одержаних результатів полягає у вдосконаленні методу автоматизації періодичних фінансових транзакцій шляхом інтеграції механізму асинхронних черг повідомлень та безсерверних обчислювачів, що, на відміну від класичних підходів, дозволяє повністю усунути витрати на простій обчислювальних потужностей (idle resources) та забезпечити гарантовану доставку повідомлень.

Практичне значення одержаних результатів.

Розроблено діючий прототип системи, який забезпечує повний цикл генерації та відправки інвойсів. Запропоноване архітектурне рішення може бути використане малими та середніми підприємствами для автоматизації бухгалтерії без необхідності інвестування у власні фізичні сервери або оренду віртуальних машин.

Особистий внесок здобувача.

1. Проведено аналіз ефективності використання Serverless-технологій для задач білінгу.
2. Розроблено архітектуру системи на базі AWS Lambda, SQS та S3.
3. Проведено експериментальне тестування системи та оцінку її економічної ефективності.

Публікації. За результатами наукових досліджень, проведених у магістерській роботі, підготовлено наукову роботу для участі у Міжнародній науковій конференції «Інформаційне суспільство: технологічні, економічні та технічні аспекти становлення» (випуск 105), яка відбулась 11-12 грудня 2025 року.

Структура магістерської роботи. Магістерська робота складається зі вступу, трьох розділів, висновків, списку використаних джерел (40 найменувань). Загальний обсяг роботи становить 91 сторінок, вона містить 23 рисунки та 3 таблиці.

РОЗДІЛ 1
АНАЛІЗ ПРОБЛЕМИ ТА ТЕОРЕТИЧНІ ОСНОВИ АВТОМАТИЗАЦІЇ ФІНАНСОВИХ ПРОЦЕСІВ

1.1. Аналіз фінансових процесів в організаціях та проблем їх автоматизації

1.1.1. Сутність та етапи процесу білінгу в сучасних організаціях

Фінансова діяльність будь-якої сучасної організації є складним комплексом взаємопов'язаних процесів, спрямованих на забезпечення обліку ресурсів, контроль грошових потоків та формування звітності. Серед широкого спектру фінансових операцій особливе місце займає білінг — процес виставлення рахунків (інвойсів) за надані послуги або виконані роботи [3]. У контексті проєктно-орієнтованих організацій або IT-компаній цей процес зазвичай включає збір інформації про відпрацьований час співробітників, прив'язку цих даних до конкретних проєктів та відділів, розрахунок сум до сплати на основі погодинних ставок та безпосередню генерацію платіжних документів у форматі PDF з подальшою їх відправкою контрагентам або співробітникам.

1.1.2. Проблематика ручної обробки даних

Традиційна організація цього процесу часто страждає від низки суттєвих недоліків. У багатьох компаніях збір даних про відпрацьовані людино-години (Workdays) все ще відбувається у напівавтоматичному режимі або за допомогою розрізнених електронних таблиць. Це призводить до розсинхронізації даних, коли інформація про співробітника, його приналежність до відділу (Department) або проєкту (Project) може бути застарілою. Ручна обробка таких даних бухгалтерією неминуче призводить до помилок людського фактора, що у фінансовій сфері може мати юридичні наслідки або призвести до репутаційних втрат. Крім того, відсутність централізованої системи ускладнює контроль, аудит і аналітику, що знижує прозорість процесів та унеможливлює своєчасне управлінське реагування.

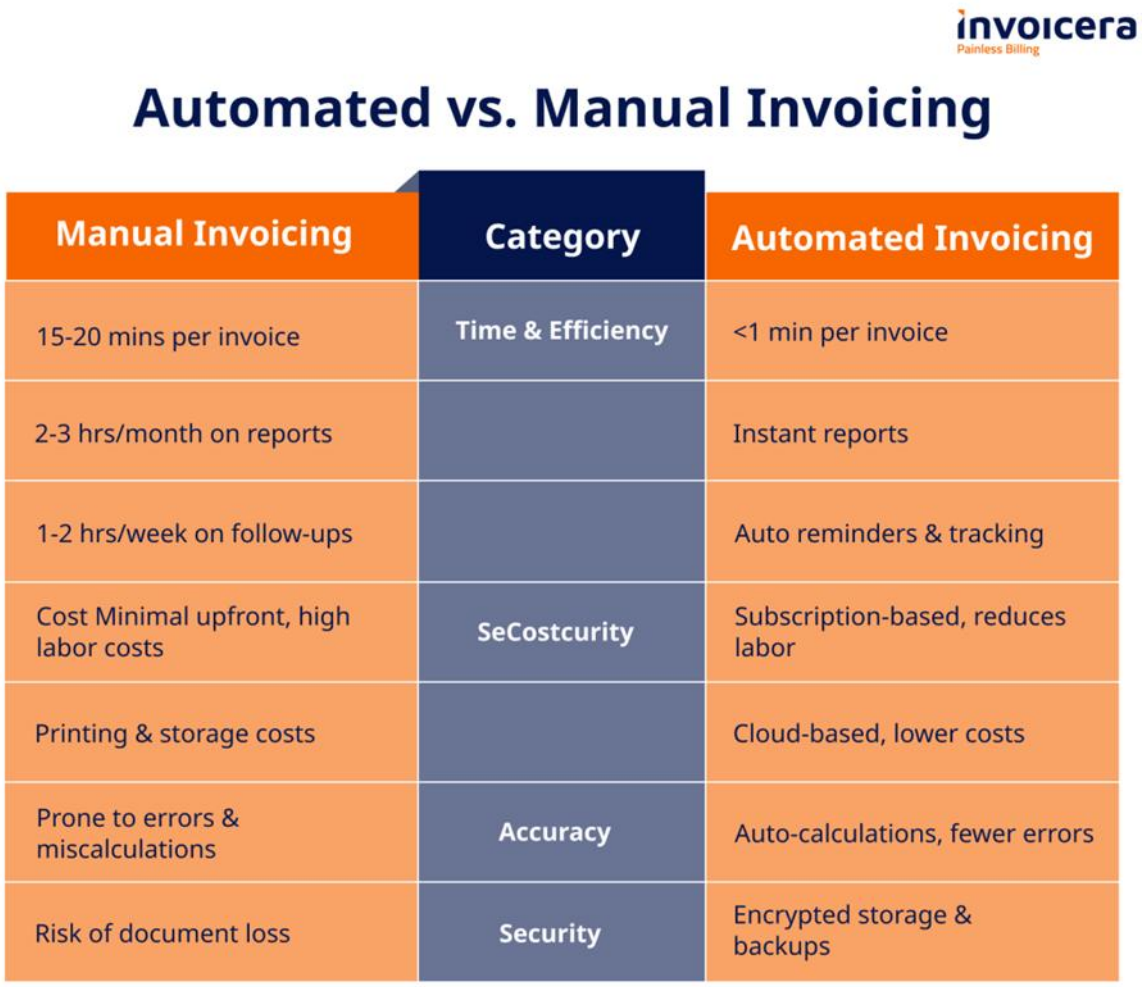


Рис. 1.1. Графік порівняння ручного та автоматичного інвойсингу

1.1.3. Специфіка навантаження та ефективність використання ресурсів

Окремою, і, можливо, найбільш критичною проблемою автоматизації білінгу є специфічний характер навантаження на інформаційну систему. На відміну від, наприклад, систем електронної комерції, де потік користувачів є відносно постійним, фінансові процеси мають яскраво виражену циклічність. Генерація інвойсів зазвичай відбувається в чітко визначені часові проміжки — наприклад, в останній день місяця або в перші дні нового звітного періоду. У цей короткий проміжок часу системі необхідно обробити великий масив даних: агрегувати інформацію про всі робочі дні всіх співробітників, згенерувати сотні або тисячі PDF-файлів та розіслати їх адресатам.

При використанні класичної монолітної архітектури, розгорнутої на виділених серверах або віртуальних машинах, виникає дилема ефективності використання

ресурсів. Для того, щоб система могла впоратися з піковим навантаженням у кінці місяця та швидко згенерувати звіти, організація змушена орендувати потужні сервери. Однак, після завершення періоду білінгу, ці потужності залишаються незадіяними протягом наступних 29-30 днів, продовжуючи споживати бюджет компанії. Ця проблема "холостого ходу" (idle resources) є ключовим бар'єром для економічної ефективності традиційної автоматизації [4]. Сервер працює постійно, споживає електроенергію та потребує обслуговування, навіть коли жоден інвойс не генерується.

1.1.4. Архітектурні обмеження: надійність та масштабованість

Крім того, синхронна обробка фінансових операцій у монолітних системах створює ризики для надійності. Якщо процес генерації інвойсів відбувається в одному потоці з вебсервером, обробка великої кількості запитів може призвести до тайм-аутів та недоступності системи для користувачів. Наприклад, помилка при генерації одного PDF-файлу або збій поштового сервера при відправці може заблокувати весь ланцюжок обробки, вимагаючи ручного втручання для повторного запуску процесу. Це підкреслює необхідність переходу від синхронних викликів до асинхронної моделі взаємодії компонентів, де ініціація процесу (запит на генерацію) відокремлена від його фактичного виконання (створення файлу та відправка) [5].

Також варто зазначити проблеми масштабованості. Зростання компанії, збільшення кількості співробітників та проєктів призводить до лінійного, а іноді й експоненційного зростання часу на генерацію звітності. У традиційних системах вертикальне масштабування (додавання потужності серверу) має фізичні та фінансові межі і часто вимагає зупинки сервісу для модернізації. Горизонтальне масштабування моноліту є складним завданням, що вимагає налаштування балансувальників навантаження та вирішення проблем узгодженості даних.

Отже, аналіз показує, що основні проблеми автоматизації фінансових процесів лежать не стільки в площині алгоритмів розрахунку, скільки в площині архітектурної організації обчислень. Існує нагальна потреба у рішеннях, які здатні динамічно виділяти ресурси саме в момент виникнення потреби (on-demand), забезпечувати ізоляцію процесів для підвищення відмовостійкості та гарантувати цілісність даних

при високому навантаженні. Це обумовлює необхідність розгляду хмарних та безсерверних технологій як основи для побудови сучасних фінансових систем.

1.2. Особливості використання хмарних технологій для фінансових систем

1.2.1. Еволюція моделей хмарних послуг та рівні абстракції

Сучасна парадигма побудови корпоративних інформаційних систем характеризується стрімкою міграцією від локальних обчислювальних потужностей (On-Premises) до хмарних середовищ (Cloud Computing). Для фінансового сектору та систем білінгу цей перехід обумовлений не лише технологічними перевагами, а й необхідністю оптимізації операційних витрат та підвищенням рівня доступності сервісів.

Використання хмарних технологій у фінансових системах має низку специфічних особливостей, які визначають вибір архітектури та інструментарію [6].

У контексті автоматизації фінансових процесів доцільно розглядати еволюцію моделей хмарних послуг від IaaS до FaaS, оскільки це безпосередньо впливає на рівень абстракції та зону відповідальності розробника.

- IaaS (Infrastructure as a Service): Надає віртуалізовані обчислювальні ресурси. Хоча це дає повний контроль над ОС, для періодичних фінансових задач це рішення є надлишковим, оскільки вимагає постійного адміністрування та оплати за час роботи віртуальної машини, навіть коли вона не виконує корисних обчислень.
- PaaS (Platform as a Service): Дозволяє розробникам фокусуватися на коді, делегуючи управління інфраструктурою провайдеру (наприклад, Azure App Service або AWS Elastic Beanstalk).
- FaaS (Function as a Service) / Serverless: Ця модель є найбільш революційною для задач білінгу. Вона дозволяє виконувати фрагменти коду (функції) у відповідь на події без необхідності управління серверами [7]. Це забезпечує автоматичне масштабування, оплату виключно за фактичне виконання функцій

та значне зниження операційних витрат, що є критично важливим для систем з нерівномірним або подієвим навантаженням.

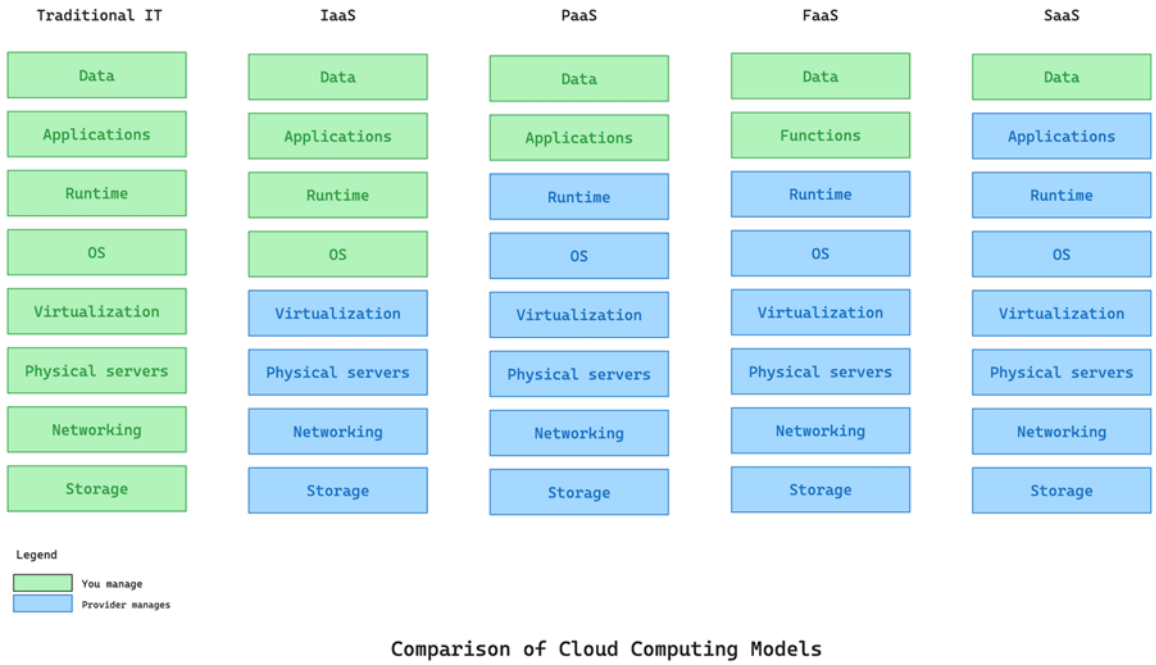


Рис. 1.2. Графік Порівняння моделей хмарних обчислень

1.2.2. Економічна ефективність та оптимізація витрат (CapEx vs OpEx)

Для систем генерації інвойсів, які характеризуються імпульсним характером навантаження (burst workloads), FaaS є оптимальним вибором. Вона дозволяє уникнути проблеми "надлишкового резервування" (over-provisioning), коли потужності закуповуються з розрахунком на пікове навантаження, але простоюють 90% часу.

Однією з головних особливостей хмарних фінансових систем є перехід від капітальних витрат (CapEx) до операційних (OpEx). У традиційній моделі організація змушена інвестувати значні кошти в закупівлю серверів "на виріст". У хмарній моделі, особливо при використанні безсерверної архітектури (наприклад, AWS Lambda), застосовується принцип Pay-as-you-go (оплата за використання) [8].

Для системи білінгу це означає, що вартість інфраструктури прямо пропорційна кількості згенерованих інвойсів. Якщо в певному місяці генерація не

проводилась, витрати на обчислювальні потужності дорівнюють нулю. Це критично важливо для малого та середнього бізнесу, де утримання власного дата-центру є економічно невиправданим.

1.2.3. Масштабованість та забезпечення рівня обслуговування (SLA)

Фінансові процеси часто мають чіткі часові рамки виконання (SLA). Наприклад, всі рахунки мають бути виставлені до 1-го числа місяця. Хмарні технології забезпечують автоматичне горизонтальне масштабування. Якщо система отримує запит на обробку 10 000 транзакцій одночасно, хмарний провайдер автоматично виділяє необхідну кількість екземплярів сервісу (instances), а після завершення обробки — звільняє їх [9]. Це гарантує стабільну продуктивність незалежно від обсягу даних, що є важкодосяжним у локальній інфраструктурі. Крім того, такий підхід дозволяє уникнути перевантажень системи у пікові періоди та забезпечити дотримання визначених SLA без надлишкового резервування ресурсів.

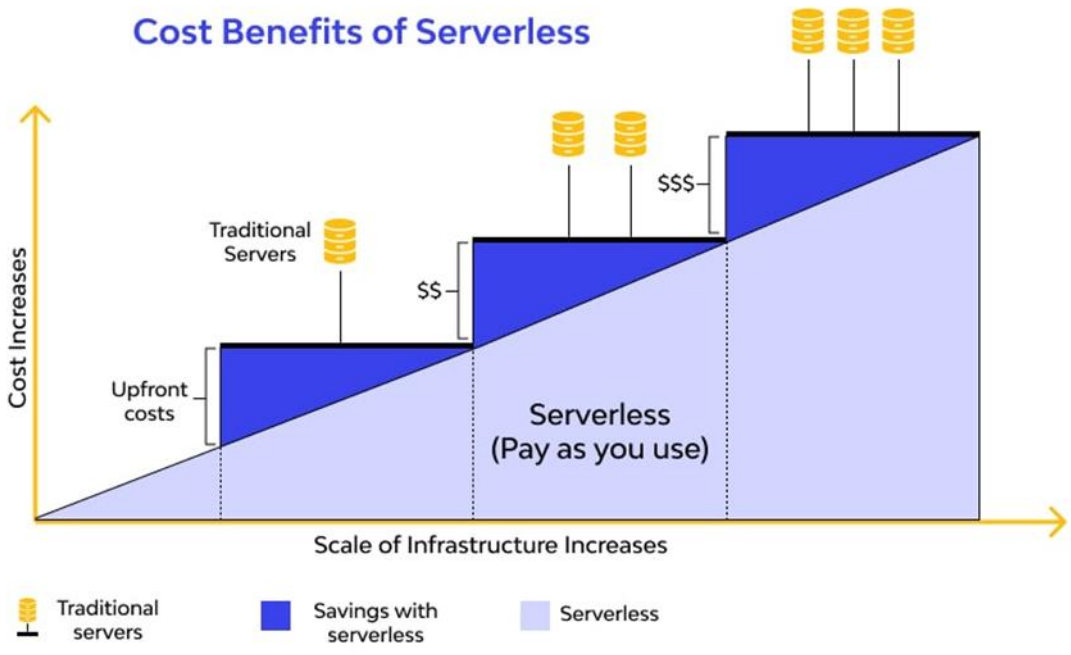


Рис. 1.3. Графік Порівняння моделей хмарних обчислень

1.2.4. Безпека даних, відповідність стандартам та відмовостійкість

Фінансові дані є чутливими, тому безпека є пріоритетом. Хмарні провайдери працюють за моделлю спільної відповідальності (Shared Responsibility Model):

- Провайдер відповідає за безпеку "хмари" (фізична безпека дата-центрів, мережева інфраструктура, захист гіпервізора).
- Клієнт відповідає за безпеку "в хмарі" (шифрування даних, управління доступом IAM, конфігурація мережевих екранів).

Сучасні хмарні платформи надають готові інструменти для забезпечення відповідності стандартам (PCI DSS, GDPR, ISO 27001), такі як автоматичне шифрування баз даних (encryption at rest) та захист каналів передачі даних (encryption in transit), що значно спрощує розробку захищених фінансових систем.

Хмарні бази даних (наприклад, Amazon RDS, що використовується в даній роботі) забезпечують високу доступність завдяки механізмам реплікації в різних зонах доступності (Availability Zones). Це гарантує, що навіть у випадку фізичного збою в одному дата-центрі, фінансові дані не будуть втрачені, а система продовжить функціонувати з мінімальною затримкою.

Таким чином, використання хмарних технологій, і зокрема безсерверної архітектури, дозволяє вирішити фундаментальні проблеми автоматизації фінансових процесів: нерівномірність навантаження, високу вартість простою обладнання та складність масштабування, забезпечуючи при цьому необхідний рівень безпеки та надійності.

1.3. Типові архітектурні підходи до автоматизації бізнес-процесів (SOA, мікросервіси, черги)

1.3.1. Еволюція архітектурних стилів та критерії вибору

Вибір архітектурного стилю є фундаментальним етапом проєктування програмного забезпечення, що визначає не лише структуру коду, але й атрибути якості системи: масштабованість, відмовостійкість, зручність супроводу та швидкість розгортання змін (Time-to-Market). У контексті автоматизації фінансових процесів, де критичними є надійність транзакцій та здатність витримувати пікові навантаження, еволюція архітектурних підходів пройшла шлях від монолітних рішень до розподілених систем [10].

1.3.2. Сервіс-орієнтована архітектура (SOA)

Історично першою спробою відійти від жорстко зв’язаних монолітних систем у великому бізнесі стала сервіс-орієнтована архітектура (Service-Oriented Architecture, SOA). Основна ідея SOA полягає у розбитті бізнес-логіки на окремі, грубозернисті (coarse-grained) сервіси, які взаємодіють між собою через мережу.

Характерною рисою класичної SOA є використання Enterprise Service Bus (ESB) — "шини" даних, яка виступає посередником, забезпечуючи маршрутизацію повідомлень, трансформацію протоколів та оркестрацію процесів. Хоча SOA дозволила повторно використовувати компоненти в різних частинах підприємства, вона часто страждала від надмірної складності, "вузького місця" у вигляді централізованої шини ESB та використання "важких" протоколів, таких як SOAP/XML.

1.3.3. Мікросервісна архітектура: переваги та недоліки

Мікросервісна архітектура виникла як еволюційний розвиток ідей SOA, але з акцентом на децентралізацію, автономність та легковагові протоколи взаємодії (REST API, gRPC) [11]. У цьому підході застосунок будується як набір невеликих сервісів, кожен з яких відповідає за окрему предметну область (Bounded Context) — наприклад, сервіс авторизації, сервіс інвойсів, сервіс звітності.

Основні переваги мікросервісів для фінансових систем:

1. Технологічна агностичність: Можливість писати різні сервіси на різних мовах програмування (наприклад, .NET для бекенду, Node.js для I/O операцій, Python для аналітики).

2. Ізоляція збоїв: Помилка в модулі генерації PDF не призводить до падіння всієї системи чи втрати доступу до бази даних користувачів, що підвищує загальну надійність платформи.

3. Незалежне розгортання**:** Можливість оновлювати окремі частини системи без зупинки всього сервісу дозволяє швидше впроваджувати зміни та зменшує ризики під час релізів.

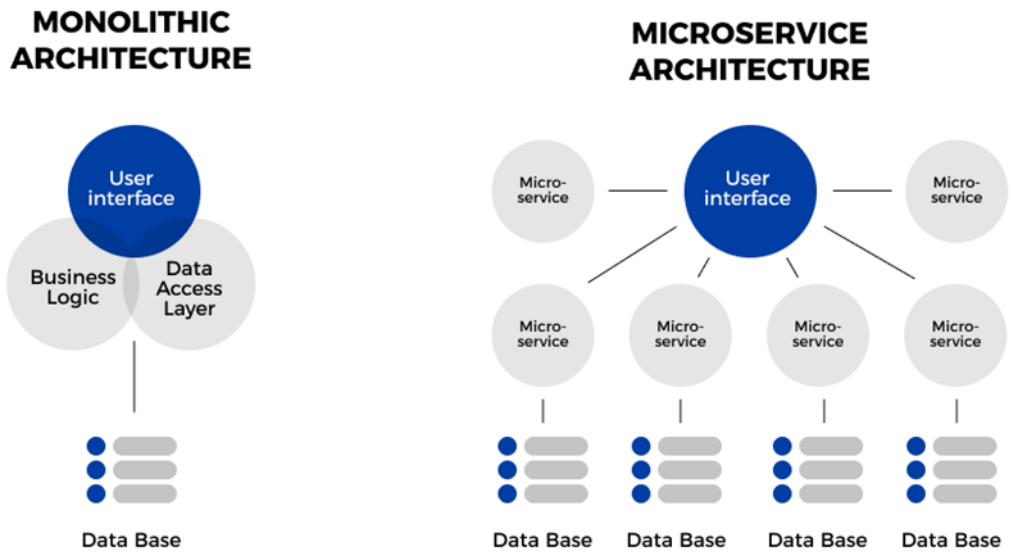


Рис. 1.4. Порівняння монолітної та мікросервісної архітектури

Однак, класична мікросервісна архітектура часто використовує синхронну взаємодію (HTTP Request/Response). Це створює проблему "ланцюгової реакції": якщо сервіс А викликає сервіс Б, а той — сервіс В, то затримка в останньому заблокує весь ланцюжок.

1.3.4. Асинхронна взаємодія та використання черг повідомлень

Для вирішення проблем синхронної комунікації та забезпечення високої пропускної здатності використовується патерн асинхронного обміну повідомленнями. Ключовим елементом такої архітектури є брокер повідомлень або черга (Message Queue), наприклад, RabbitMQ, Apache Kafka або хмарні сервіси типу AWS SQS (Simple Queue Service).

Використання черг надає системі унікальні властивості, критичні для білінгових процесів:

1. Тимчасова розв'язка (Temporal Decoupling): Компонент-виробник (Producer) може відправити завдання на генерацію інвойсу і продовжити роботу, не чекаючи, поки компонент-споживач (Consumer/Worker) його обробить. Споживач може бути навіть тимчасово вимкнений — повідомлення збережеться в черзі.

2. Згладжування навантаження (Load Leveling): Це, мабуть, найважливіша перевага для фінансових систем із періодичним навантаженням. Якщо в систему одночасно надходить 10 000 запитів на генерацію документів, вони не "кладуть" базу даних, а стають у чергу. Воркери (обробники) розбирають цю чергу з тією швидкістю, яку дозволяють ресурси, гарантуючи стабільність системи.

3. Гарантована доставка: Сучасні черги підтримують механізми підтвердження обробки (ACK). Якщо воркер падає під час обробки інвойсу, повідомлення повертається в чергу і буде оброблено іншим екземпляром воркера.

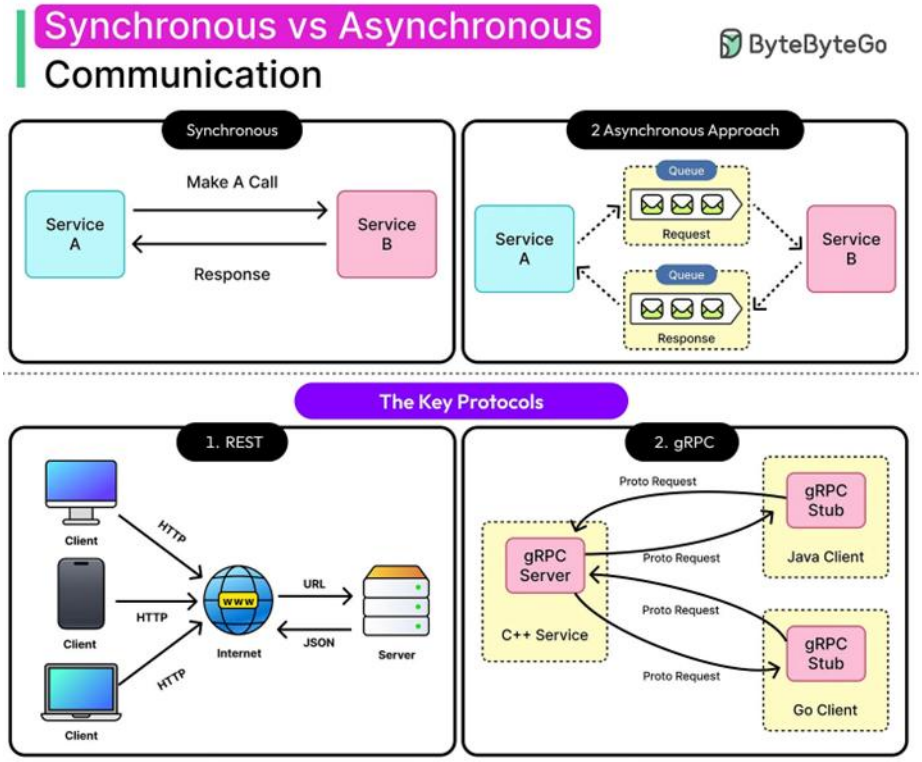


Рис. 1.5. Порівняння синхронної (HTTP) та асинхронної (Queue) взаємодії

1.3.5. Подієво-орієнтована архітектура (EDA)

Поєднання мікросервісів (або FaaS-функцій) із чергами повідомлень формує подієво-орієнтовану архітектуру. У такій системі зміни стану однієї сутності (наприклад, "Користувач натиснув кнопку 'Згенерувати звіт'") породжують подію, на яку реагують інші компоненти.

Саме цей підхід є найбільш сучасним стандартом для побудови масштабованих хмарних систем. Він дозволяє інтегрувати безсерверні обчислення

(Serverless), де функції-обробники (Lambda) "прокидаються" лише тоді, коли в черзі з'являється повідомлення, що забезпечує максимальну економічну ефективність та автоматичне масштабування.

1.4. Формулювання проблеми автоматизації фінансових процесів у хмарному середовищі

1.4.1. Протиріччя між вимогами продуктивності та економічної ефективності

Узагальнюючи результати аналізу, проведеного у попередніх підрозділах, можна констатувати наявність суттєвого розриву між вимогами сучасного бізнесу до фінансових систем та можливостями традиційних архітектурних рішень. Специфіка білінгових процесів, яка полягає у їхній циклічності та крайній нерівномірності навантаження, створює унікальний клас інженерних задач, що не мають ефективного вирішення в рамках класичної парадигми виділених серверів [13].

Ключова проблема полягає у фундаментальному протиріччі між необхідністю забезпечення високої пікової продуктивності системи та вимогою мінімізації сукупної вартості володіння (Total Cost of Ownership, TCO). Традиційні підходи до розгортання програмного забезпечення змушують архітекторів обирати між двома неефективними стратегіями:

- Стратегія надлишкового резервування: Передбачає закупівлю потужностей під пікове навантаження, що призводить до простою ресурсів у періоди між звітними датами та економічної неефективності.
- Стратегія орієнтації на середнє навантаження: Несе в собі ризики відмови в обслуговуванні (Denial of Service) у критичні моменти закриття фінансового періоду, коли потік транзакцій різко зростає [14].

1.4.2. Технологічні обмеження синхронної обробки даних

Окрім економічного аспекту, існують суттєві технологічні обмеження, пов'язані з надійністю синхронної обробки даних. У монолітних або тісно зв'язаних

мікросервісних системах процес генерації інвойсів часто реалізується як послідовний ланцюжок синхронних викликів: запит до бази даних, формування PDF-файлу, звернення до поштового сервісу. Така архітектура є вразливою до точкових збоїв: тимчасова недоступність зовнішнього SMTP-сервера або помилка при рендерингу одного документа може призвести до каскадного збою всього процесу масової розсилки. Відсутність вбудованих механізмів гарантованої доставки та повторної обробки помилок у синхронних системах змушує розробників імплементувати складну логіку компенсаційних транзакцій, що ускладнює підтримку коду та знижує загальну стабільність рішення [15].

Також варто зазначити проблему масштабованості компонентів системи, що мають різну природу споживання ресурсів. Генерація PDF-документів є процесоромісткою операцією (CPU-bound), тоді як відправка електронних листів або запис у базу даних є операціями введення-виведення (I/O-bound). У класичній архітектурі, де ці функції об'єднані в одному додатку, неможливо масштабувати їх незалежно. Це призводить до нераціонального використання ресурсів, коли для прискорення генерації документів доводиться дублювати і ті компоненти, які не потребують додаткової потужності [16].

1.4.3. Постановка науково-технічної задачі

Отже, науково-технічна проблема, що вирішується в даній роботі, формулюється як необхідність розробки методу автоматизації періодичних фінансових процесів, який забезпечував би динамічну адаптацію обчислювальних ресурсів до поточного навантаження в реальному часі, гарантував би цілісність та збереження даних при асинхронній обробці транзакцій, та дозволяв би повністю усунути витрати на інфраструктуру в періоди простою системи. Вирішення цієї проблеми лежить у площині переходу від ресурсо-орієнтованої до подієво-орієнтованої моделі архітектури (Event-Driven Architecture) з використанням безсерверних обчислень (Serverless).

Для досягнення поставленої мети та розв'язання сформульованої проблеми в роботі необхідно вирішити наступні завдання:

1. Розробити формальну модель предметної області, що включає опис сутностей фінансового обліку, станів документів та логіку їх трансформації, а також визначити вимоги до ізоляції даних та консистенції транзакцій.

2. Обґрунтувати та спроєктувати архітектуру системи на базі хмарної платформи AWS, яка поєднує безсерверні обчислювальні модулі (AWS Lambda) для виконання бізнес-логіки та керовані черги повідомлень (AWS SQS) для забезпечення асинхронної взаємодії та згладжування пікових навантажень.

3. Реалізувати програмний прототип системи, що демонструє повний цикл автоматизації білінгу: від збору даних про відпрацьований час до генерації інвойсів у форматі PDF, їх збереження у хмарному сховищі S3 та відправки кінцевим користувачам.

4. Провести експериментальне дослідження розробленої системи, оцінити її поведінку під навантаженням, перевірити коректність роботи механізмів масштабування та провести порівняльний аналіз економічної ефективності запропонованого рішення у порівнянні з традиційними серверними архітектурами.

1.5. Висновки до розділу

У даному розділі проведено комплексний аналіз предметної області автоматизації фінансових процесів та досліджено сучасні технологічні підходи до вирішення проблем масштабованості та економічної ефективності корпоративних систем.

За результатами виконаних досліджень можна зробити наступні висновки:

1. Аналіз бізнес-процесів показав, що процедура білінгу та генерації звітності характеризується яскраво вираженою циклічністю та нерівномірністю навантаження (імпульсний характер). Традиційні методи організації цих процесів, що базуються на ручній обробці даних або використанні монолітних систем, мають суттєві недоліки: високий ризик помилок, низьку швидкість реакції на зміни та нераціональне використання ресурсів.

2. Дослідження моделей розгортання виявило фундаментальну проблему традиційної серверної інфраструктури (On-Premises та IaaS) — проблему «холостого ходу» (idle resources). Утримання виділених серверів, розрахованих на пікове навантаження, у періоди затишшя є економічно невиправданим. Встановлено, що перехід до моделі FaaS (Function-as-a-Service) та безсерверних обчислень дозволяє трансформувати структуру витрат з капітальних (CapEx) на операційні (OpEx), забезпечуючи оплату лише за фактично використаний час обчислень.

3. Огляд архітектурних стилів підтвердив обмеженість синхронної взаємодії (HTTP Request-Response) для побудови високонавантажених фінансових систем через ризики каскадних збоїв. Обґрунтовано доцільність використання подієво-орієнтованої архітектури (Event-Driven Architecture) з використанням черг повідомлень. Цей підхід забезпечує тимчасову розв'язку компонентів, гарантовану доставку повідомлень та згладжування пікових навантажень.

4. Сформульовано науково-технічну проблему, яка полягає у протиріччі між необхідністю забезпечення високої пікової продуктивності системи білінгу та вимогою мінімізації витрат на її експлуатацію. Вирішення цієї проблеми лежить у площині створення адаптивної системи, здатної до автоматичного масштабування від нуля до необхідного максимуму без втручання адміністратора.

Отримані результати аналізу створюють теоретичне підґрунтя для наступного етапу роботи — розробки математичних моделей, проєктування архітектури та програмної реалізації системи, що буде розглянуто у другому та третьому розділах дисертації.

РОЗДІЛ 2
ФОРМАЛЬНЕ РІШЕННЯ ПРОБЛЕМИ ТА МОДЕЛЮВАННЯ СИСТЕМИ

2.1. Узагальнена модель системи автоматизації фінансових процесів

2.1.1. Обґрунтування вибору архітектурного стилю

Для реалізації системи автоматизації білінгу обрано модель розподіленої безсерверної системи (Serverless Architecture). Така модель передбачає відмову від монолітного ядра на користь набору незалежних функцій, які взаємодіють між собою через події.

Узагальнена структура системи складається з чотирьох логічних блоків: блоку взаємодії з користувачем (API), блоку асинхронної обробки (Workers), блоку повідомлень (Notifications) та рівня зберігання даних (Storage).

2.1.2. Структурна модель компонентів системи

Система побудована на базі хмарних сервісів AWS та складається з наступних ключових елементів:

1. Точка входу (Web API Lambda): Основний компонент, який приймає HTTP-запити від клієнтського інтерфейсу (Frontend). Його задача — валідація вхідних даних, авторизація користувача та збереження початкової інформації в базу даних. Цей компонент працює синхронно, забезпечуючи миттєву відповідь користувачу.
2. Черги повідомлень (AWS SQS): Використовуються як буфер між компонентами для забезпечення асинхронності. У системі виділено дві окремі черги:
   - ◦ Worker SQS Queue — для задач генерації документів;
   - ◦ Emailer SQS Queue — для задач відправки листів.
3. Обчислювальні вузли (Worker & Emailer Lambdas): Функції, що виконують "важку" роботу у фоновому режимі. Worker Lambda відповідає за формування PDF-файлів, а Emailer Lambda — за їх доставку кінцевим отримувачам.
4. Сховища даних:

◦ RDS Postgres Database — реляційна база даних для зберігання структурованої інформації (користувачі, проєкти, метадані інвойсів) [17].

◦ Amazon S3 — об'єктне сховище для зберігання згенерованих бінарних файлів (PDF).

2.1.3. Функціональна модель потоків даних (Data Flow)

Взаємодія компонентів системи відбувається за чітко визначеним алгоритмом, який забезпечує цілісність даних та відмовостійкість. Процес обробки фінансового документа проходить наступні етапи:

- Ініціалізація процесу. Користувач надсилає запит на створення інвойсу до Web API Lambda. Система валідує дані, зберігає метаінформацію в RDS Postgres та публікує повідомлення invoice_id у чергу Worker SQS Queue, миттєво повертаючи відповідь успіху клієнта (HTTP 202 Accepted).
- Генерація документу. Worker Lambda автоматично активується при появі повідомлення в черзі. Компонент отримує необхідні дані з БД, виконує бізнес-розрахунки, генерує PDF-файл та завантажує його в Amazon S3. Після успішного завантаження оновлюється запис у БД (додається invoiceUrl) і відправляється сповіщення в Emailer SQS Queue.
- Дистрибуція та нотифікація. Emailer Lambda обробляє повідомлення з черги розсилки. Функція завантажує згенерований файл зі сховища S3 та відправляє електронний лист із вкладенням на адресу співробітника.
- Отримання результатів. Для перегляду документу через вебінтерфейс, Web API Lambda генерує тимчасове підписане посилання (Pre-signed URL) до об'єкта в S3, що дозволяє користувачу безпечно завантажити файл без відкриття публічного доступу до сховища [18].

Така модель дозволяє чітко розділити зони відповідальності: API відповідає лише за прийом заявок, а важкі операції генерації та розсилки винесені у фонові процеси, що дозволяє системі легко масштабуватися при великій кількості одночасних запитів.

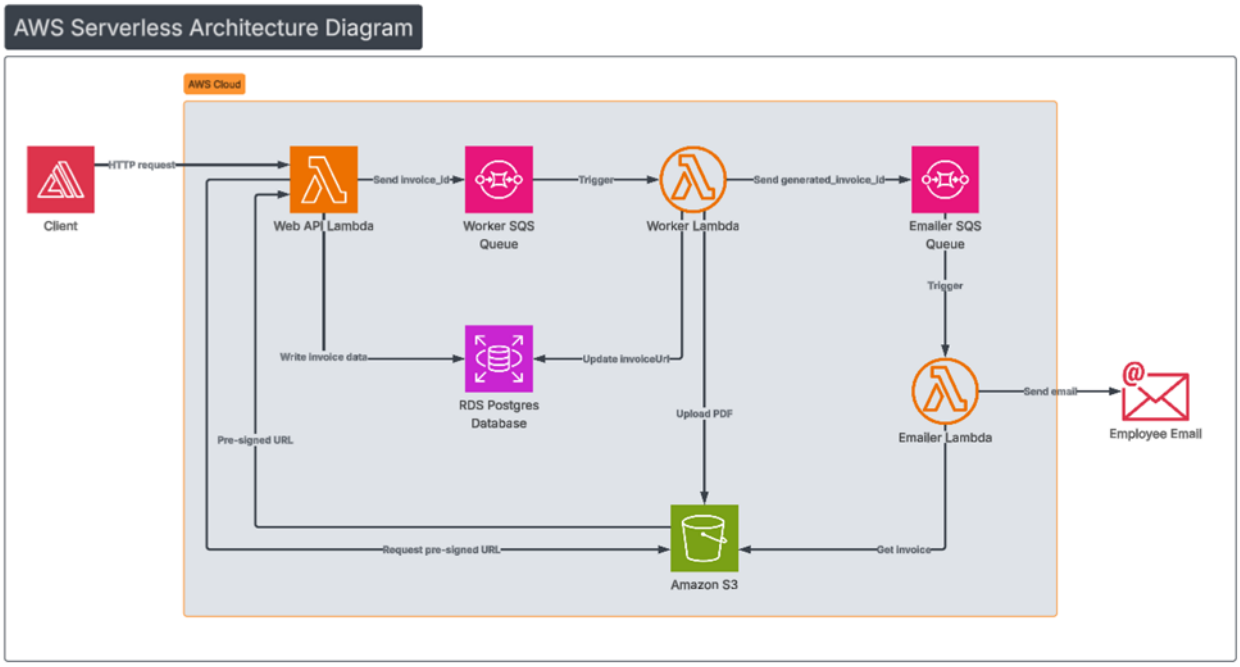


Рис. 2.1. Узагальнена структурна схема та діаграма потоків даних (Data Flow) системи.

2.2. Формалізація функціональних та нефункціональних вимог

Визначення вимог є критичним етапом проєктування програмної системи, що дозволяє окреслити межі системи, критерії її успішної реалізації та забезпечити узгодженість між очікуваннями замовника і можливостями розробників. Згідно зі стандартом IEEE 830-1998, специфікація вимог до програмного забезпечення повинна бути однозначною, повною, верифікованою та модифікованою. Для системи вимоги класифікуються на дві основні категорії: функціональні вимоги, які описують бізнес-логіку та поведінку системи, та нефункціональні вимоги, які визначають атрибути якості системи відповідно до моделі ISO/IEC 25010 [19].

Методологія формалізації вимог у цій роботі базується на структурованому підході з використанням унікальних ідентифікаторів для кожної вимоги. Функціональні вимоги позначаються префіксом FR (Functional Requirement), а нефункціональні — NFR (Non-Functional Requirement). Такий підхід забезпечує простоту трасування вимог на етапах проєктування, імплементації та тестування

системи, що особливо важливо для магістерського дослідження, яке передбачає повний цикл розробки від концепції до реалізації.

2.2.1. Функціональні вимоги

Функціональні вимоги описують те, що система повинна робити — конкретні функції, операції та сервіси, які вона надає користувачам. Система призначена для внутрішнього використання в організації, тому основний акцент робиться на адміністративних функціях та автоматизації рутинних операцій, пов'язаних з обліком робочого часу та генерацією фінансової документації. Функціонал системи декомпозується за основними сутностями предметної області відповідно до принципів domain-driven design, що дозволяє забезпечити природну відповідність між бізнес-процесами та програмною архітектурою.

Забезпечення безпеки доступу до системи є першочерговою вимогою для будь-якого корпоративного програмного забезпечення. Підсистема автентифікації та авторизації відповідає за ідентифікацію користувачів та контроль їхніх прав доступу до ресурсів системи.

Система повинна надавати механізм автентифікації користувачів за логіном та паролем (FR-1.1). Враховуючи внутрішній характер використання системи та обмежену кількість користувачів, реєстрація нових облікових записів через вебінтерфейс не передбачається. Натомість, створення облікових записів виконується адміністратором безпосередньо на рівні бази даних (FR-1.2), що забезпечує додатковий контроль над доступом до системи та відповідає політиці інформаційної безпеки організації. Доступ до програмного інтерфейсу (API) повинен бути захищений таким чином, щоб неавтентифіковані запити автоматично відхилялися системою (FR-1.3), що запобігає несанкціонованому доступу до конфіденційних даних.

Ефективне функціонування системи обліку робочого часу неможливе без підтримки довідкової інформації, яка описує організаційну структуру компанії та її проєктну діяльність. Система повинна забезпечувати повний цикл операцій створення, читання, оновлення та видалення даних (CRUD) для всіх адміністративних сутностей.

Модуль управління профілями співробітників (FR-2.1) повинен зберігати всю інформацію, необхідну для ідентифікації працівника та забезпечення комунікації з ним, включаючи персональні дані та контактну інформацію. Ця функціональність є критичною для наступних етапів обробки інвойсів, оскільки саме на основі цих даних формуються персоналізовані документи.

Реєстр проєктів (FR-2.2) дозволяє вести облік усіх проєктів, над якими працює організація. Це забезпечує можливість деталізованого аналізу розподілу робочого часу між різними напрямками діяльності компанії. Аналогічно, модуль управління організаційною структурою (FR-2.3) підтримує ієрархію відділів компанії, що є необхідним для звітності та аналізу продуктивності на рівні підрозділів.

Центральною функціональною можливістю системи є детальний облік відпрацьованого часу співробітників. Система повинна надавати інтерфейс для внесення записів про робочі дні з обов'язковою прив'язкою до конкретного співробітника (FR-3.1). Кожен запис містить інформацію про дату, кількість годин та проєкт, в рамках якого виконувалася робота.

Важливою вимогою є забезпечення можливості перегляду, редагування та видалення записів про робочий час (FR-3.2). Це необхідно для корекції помилкових або неповних даних перед генерацією офіційної звітності та фінансових документів. Можливість внесення коректив на етапі до формування інвойсів значно знижує ризик помилок у фінальних документах та підвищує загальну якість облікових процесів.

Модуль управління інвойсами реалізує основну бізнес-логіку системи та є найбільш складним з функціональної точки зору. Цей модуль інтегрує дані з усіх попередніх підсистем для створення комплексного фінансового документу.

Система повинна надавати функціонал створення нової сутності інвойсу (FR-4.1), який ініціює складний багатоетапний процес обробки. Критичною архітектурною вимогою є забезпечення асинхронної обробки: при успішному створенні запису про інвойс у базі даних система повинна автоматично ініціювати асинхронний процес генерації документу шляхом публікації повідомлення в чергу обробки (FR-4.2). Така архітектура забезпечує чутливість користувацького інтерфейсу,

оскільки операція створення інвойсу не блокується тривалим процесом генерації PDF-файлу [20].

Система повинна забезпечувати повний спектр операцій перегляду: як список усіх інвойсів з можливістю фільтрації та сортування, так і детальну інформацію про конкретний інвойс (FR-4.3). Функція генерації PDF-документу (FR-4.4) повинна автоматично агрегувати дані про співробітника, його робочі дні за відповідний період та формувати документ згідно з встановленим шаблоном. Завершальним етапом є автоматична відправка згенерованого інвойсу на електронну адресу співробітника (FR-4.5), що повністю автоматизує процес розповсюдження фінансової документації. Такий підхід мінімізує ручне втручання, знижує ймовірність помилок та забезпечує своєчасне отримання фінансових документів усіма зацікавленими сторонами.

У таблиці 2.1 наведено зведену інформацію про всі функціональні вимоги до системи з метою спрощення їх трасування на наступних етапах розробки.

Таблиця 2.1

Зведена таблиця функціональних вимог

Ідентифікатор	**Категорія**	**Опис вимоги**	**Метрика верифікації**
NFR-1.1	Масштабованість	Автоматичне горизонтальне масштабування обробників	Час обробки черги при навантаженні
NFR-1.2	Продуктивність	Незалежність часу відповіді API від обробки	Час відповіді < 500 мс
NFR-2.1	Надійність	Гарантована доставка повідомлень з retry	Відсоток втрачених повідомлень = 0%
NFR-2.2	Надійність	Узгодженість стану БД та генерації файлів	Відсутність сиріт-файлів
NFR-3.1	Безпека	Доступ до файлів через підписані посилання	Неможливість прямого доступу
NFR-3.2	Безпека	Захист конфіденційних даних	Відсутність секретів у коді

2.2.2. Нефункціональні вимоги

Нефункціональні вимоги визначають не те, що система робить, а те, як саме вона це робить. Відповідно до стандарту ISO/IEC 25010, нефункціональні характеристики включають продуктивність, надійність, безпеку, супроводжуваність та інші атрибути якості системи [2]. Для хмарної безсерверної архітектури, обраної для реалізації системи, особливо важливими є вимоги до масштабованості, відмовостійкості та безпеки.

Однією з ключових переваг безсерверної архітектури є природна здатність до горизонтального масштабування без необхідності ручного втручання адміністраторів. Система повинна підтримувати автоматичне горизонтальне масштабування для компонентів обробки повідомлень (NFR-1.1), зокрема для функцій генерації PDF-документів та відправки електронної пошти. При зростанні черги необроблених повідомлень платформа повинна автоматично збільшувати кількість паралельних обробників, а при зниженні навантаження — зменшувати їх кількість до мінімуму, що забезпечує оптимальне використання обчислювальних ресурсів та мінімізацію витрат.

Критичною вимогою до продуктивності є забезпечення незалежності часу відповіді API від тривалості обробки запиту (NFR-1.2). Операція створення інвойсу через програмний інтерфейс повинна завершуватися негайно після успішного збереження метаданих у базі даних, не очікуючи на завершення процесу генерації PDF-файлу. Це досягається завдяки асинхронній архітектурі з використанням черг повідомлень та забезпечує прийнятний користувацький досвід навіть при високому навантаженні на систему обробки документів.

Надійність є критичним атрибутом якості для системи обліку робочого часу, оскільки втрата даних про згенеровані інвойси може призвести до фінансових розбіжностей та юридичних проблем. Система повинна гарантувати, що жоден запит на генерацію інвойсу не буде втрачений навіть у разі тимчасових збоїв інфраструктури (NFR-2.1). Це досягається через механізм повторних спроб: якщо під час генерації PDF-документу або відправки електронної пошти виникає помилка, завдання автоматично повертається в чергу для повторної обробки. Кількість спроб та

інтервали між ними повинні бути налаштовані таким чином, щоб мінімізувати ймовірність остаточної втрати повідомлення при збереженні прийнятної затримки обробки.

Узгодженість стану системи є фундаментальною вимогою до архітектури (NFR-2.2). Генерація файлу повинна ініціюватися виключно після успішного збереження метаданих інвойсу в базі даних, що запобігає ситуації, коли PDF-документ створено, але в системі відсутня відповідна інформація про інвойс, або навпаки. Це досягається через транзакційний підхід: спочатку виконується атомарна операція запису в базу даних, і лише після її успішного завершення публікується повідомлення в чергу обробки.

Безпека даних є пріоритетною вимогою для будь-якої системи, що обробляє конфіденційну фінансову та персональну інформацію. Прямий публічний доступ до сховища PDF-файлів повинен бути повністю заблокований (NFR-3.1). Натомість, система повинна генерувати тимчасові підписані посилання з обмеженим терміном дії, які надаються користувачам через захищений програмний інтерфейс. Такий підхід забезпечує контроль над доступом до документів та запобігає несанкціонованому розповсюдженню конфіденційної інформації.

Усі конфіденційні дані, включаючи параметри підключення до бази даних, облікові дані хмарних сервісів та ключі шифрування, не повинні зберігатися у відкритому вигляді в коді програми або конфігураційних файлах (NFR-3.2). Замість цього використовуються спеціалізовані сервіси управління секретами, які забезпечують безпечне зберігання, контроль доступу та аудит використання конфіденційної інформації. Це відповідає сучасним практикам DevSecOps та знижує ризик компрометації системи через витік облікових даних.

Таблиця 2.2

Зведена таблиця нефункціональних вимог

Ідентифікатор	**Категорія**	**Опис вимоги**	**Метрика верифікації**
NFR-1.1	Масштабованість	Автоматичне горизонтальне	Час обробки черги при навантаженні

Ідентифікатор	Категорія	Опис вимоги	Метрика верифікації
		масштабування обробників	
NFR-1.2	Продуктивність	Незалежність часу відповіді API від обробки	Час відповіді < 500 мс
NFR-2.1	Надійність	Гарантована доставка повідомлень з retry	Відсоток втрачених повідомлень = 0%
NFR-2.2	Надійність	Узгодженість стану БД та генерації файлів	Відсутність сиріт-файлів
NFR-3.1	Безпека	Доступ до файлів через підписані посилання	Неможливість прямого доступу
NFR-3.2	Безпека	Захист конфіденційних даних	Відсутність секретів у коді

Формалізація вимог до системи виконана відповідно до міжнародних стандартів IEEE 830-1998 та ISO/IEC 25010, що забезпечує повноту, однозначність та верифікованість специфікації. Виділено 13 функціональних вимог, що охоплюють п'ять основних підсистем: автентифікацію, управління довідниками, облік робочого часу та управління інвойсами. Нефункціональні вимоги сформульовані з урахуванням обраної безсерверної хмарної архітектури та акцентують увагу на масштабованості, надійності та безпеці системи. Структурований підхід до опису вимог з використанням унікальних ідентифікаторів та зведених таблиць забезпечує зручність трасування вимог на всіх етапах життєвого циклу розробки програмного забезпечення.

2.3. Математичне та логічне описання бізнес-процесів

2.3.1. Модель даних системи

Систему S можна представити як сукупність трьох основних списків (масивів даних), з якими відбуваються операції:

$$S=E,W,I \tag{2.1}$$

де:

1. E — список співробітників (Employees);
2. W — список записів про робочі години (Workdays);
3. I — список інвойсів (Invoices).

Кожен запис про робочий час w містить три ключові параметри: дату (d), кількість годин (h) та посилання на співробітника (i_{demp}):

$$w=\left(d, h, id_{emp}\right) \tag{2.2}$$

2.3.2. Формули розрахунку заробітної плати

Алгоритм розрахунку суми інвойсу базується на визначенні вартості однієї години роботи співробітника в конкретному місяці:

Нехай задано місяць M та рік Y:

- Крок 1: Спочатку визначається кількість робочих будніх днів (понеділок–п'ятниця) у заданому місяці. Позначимо цю кількість як D_{work}. Нормативна кількість годин H_{norm} розраховується виходячи зі стандартного 8-годинного робочого дня:

$$H_{norm}=D_{work} \times 8 \tag{2.3}$$

- Крок 2: Розрахунок вартості години роботи (Rate). Нехай S_{month} — це фіксована місячна зарплата співробітника, яка визначена у відповідності до його контракту або посадового окладу. Щоб визначити реальну вартість однієї години роботи R_{hour} у цьому місяці, необхідно врахувати фактичну кількість відпрацьованих днів та стандартну тривалість робочого дня, що дозволяє більш точно оцінити витрати на робочу силу для бухгалтерського обліку та білінгових розрахунків.

$$R_{hour}=\frac{S_{month}}{H_{norm}} \tag{2.4}$$

- Крок 3: Розрахунок відпрацьованого часу. Система сумує всі години з записів w, які належать цьому співробітнику та потрапляють у даний місяць. Загальний час H_{total}:

$$H_{total}=\sum h_i \tag{2.5}$$

де h_i — години в окремий день.

- Крок 4: Підсумкова сума до виплати Sum розраховується як добуток відпрацьованого часу на вартість години з округленням до 2 знаків:

$$\sum = H_{total} \times R_{hour} \tag{2.6}$$

2.3.3. Логіка валідації (перевірки даних)

Перед створенням інвойсу дані проходять перевірку. Результат перевірки $IsValid$ може бути «Істина» (True) або «Хибність» (False).

Валідація інвойсу описується як логічне "I" (AND) між декількома умовами:

$$IsValid(Invoice)=C_{month} \wedge C_{year} \wedge C_{emp} \tag{2.7}$$

де умови визначені так:

- C_{month}: номер місяця має бути від 1 до 12 ($1 \leq$ Month ≤ 12);
- C_{year}: рік має бути більше нуля (Year>0);
- C_{emp}: співробітник має бути вказаний (EmployeeId ≠ Null).

Якщо хоча б одна умова хибна, запит відхиляється.

2.3.4. Модель життєвого циклу інвойсу

Процес обробки інвойсу можна описати як перехід між трьома станами. Позначимо стани:

1. $Status_0$ — Створено (дані записані в БД).
2. $Status_1$ — Згенеровано (PDF-файл створено і завантажено).
3. $Status_2$ — Відправлено (Email надіслано).

Логіка переходів:

Крок 1: Система перевіряє наявність даних. Якщо дані є, запускається генерація PDF.

$$Status_0 \xrightarrow{\text{Генерація } PDF} Status_1 \tag{2.8}$$

Крок 2: Якщо файл успішно завантажено в сховище (Url ≠ Null), запускається відправка пошти.

$$Status_1 \xrightarrow{SMTP \text{ відправка}} Status_2 \tag{2.9}$$

У разі помилки на будь-якому етапі процес повторюється (Retry), не змінюючи фінальний статус на помилковий одразу.

2.4. Розроблення архітектурної моделі системи

При проєктуванні програмного комплексу було використано гібридний архітектурний підхід, що поєднує принципи «Чистої архітектури» (Clean Architecture) та архітектури вертикальних зрізів (Vertical Slice Architecture). Вибір такої стратегії обумовлений необхідністю забезпечення чіткого розділення відповідальності між компонентами системи, ізоляції бізнес-логіки від інфраструктурних залежностей та спрощення підтримки кодової бази в умовах розподіленого безсерверного середовища [21]. Крім того, архітектура вертикальних зрізів дозволяє групувати функціональність навколо конкретних бізнес-сценаріїв, що покращує читабельність і навігацію в коді. Поєднання цих підходів також спрощує тестування окремих функцій та зменшує взаємозалежність модулів.

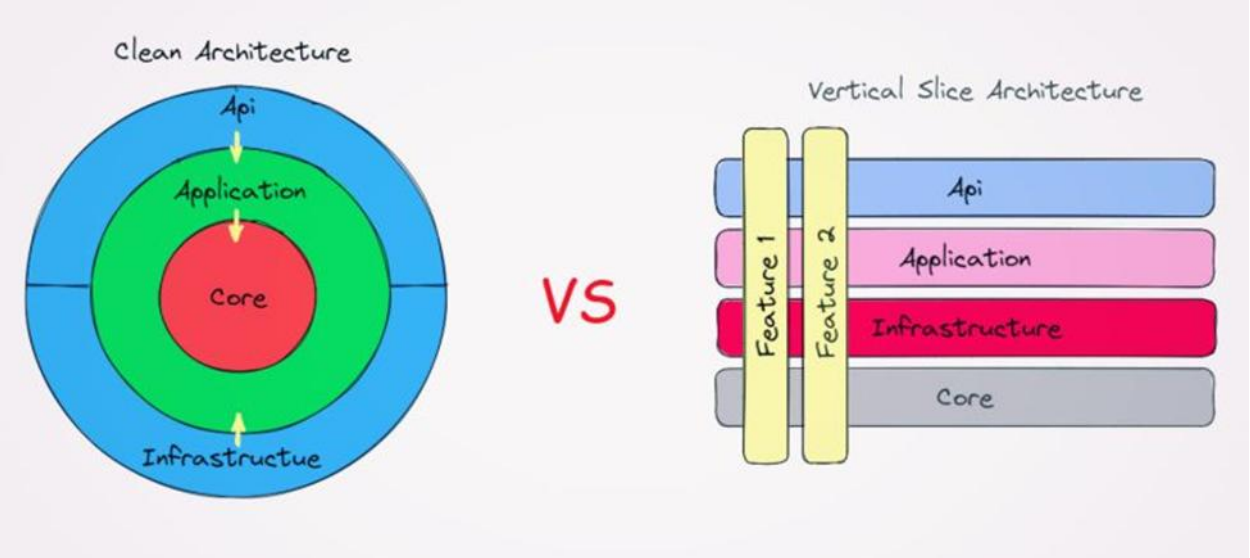


Рис. 2.2. Порівняння clean architecture і Vertical slice architecture

2.4.1. Структурна декомпозиція рішення

Архітектура рішення реалізує принцип інверсії залежностей (Dependency Inversion Principle), згідно з яким високорівневі політики не повинні залежати від низькорівневих деталей реалізації. Фізично рішення структуроване у вигляді набору взаємопов'язаних проєктів, які можна логічно об'єднати у концентричні шари.

Центральним елементом системи, що забезпечує стабільність бізнес-правил, є шар доменної моделі, представлений проєктом Domain. Цей шар є повністю автономним і не містить жодних зовнішніх залежностей. У ньому визначені основні сутності предметної області, такі як User, Employee, Invoice, Workday, Project та Department. Всі сутності наслідуються від базового класу AuditableEntity, що дозволяє уніфікувати механізми відстеження історії змін записів. Навколо доменної моделі побудовано шар ядра системи Core, який виступає сполучною ланкою між абстракціями та реалізацією. Він містить інтерфейси репозиторіїв, сервісів, контракти даних (DTO) для обміну інформацією між модулями, а також профілі мапінгу об'єктів. Саме в цьому шарі визначені абстракції інфраструктурних компонентів, таких як ISqsPublisher для роботи з чергами та IS3FileService для роботи з файловим сховищем, що дозволяє бізнес-логіці залишатися незалежною від конкретного хмарного провайдера.

Реалізація прикладної логіки зосереджена в проєкті Application. Цей шар відповідає за оркестрацію бізнес-процесів, обробку команд та запитів. Тут розміщено сервіси, такі як WebApiService, що координує виконання операцій CRUD, та SqsMessageDispatcher, який забезпечує маршрутизацію асинхронних повідомлень. Особливістю реалізації цього шару є використання допоміжних механізмів для побудови динамічних запитів, зокрема класу PredicateBuilder, який дозволяє конструювати складні дерева виразів (Expression Trees) для фільтрації даних, та фабрики MessageHandlerFactory, що інкапсулює логіку створення обробників подій.

Інфраструктурний шар, реалізований у проєкті Persistence, відповідає за збереження стану системи. Доступ до даних організовано через патерн «Репозиторій» з використанням універсальної реалізації GenericRepository, що працює поверх ORM Entity Framework Core [22]. Важливою архітектурною деталлю є використання механізму перехоплювачів (Interceptors) — SoftDeleteInterceptor та UpdateAuditableEntityInterceptor. Це дозволяє винести службову логіку, таку як «м'яке» видалення записів та автоматичне оновлення полів аудиту, за межі бізнес-транзакцій, забезпечуючи чистоту коду сервісів [23]. Окремо виділено проєкт Auth, який інкапсулює логіку безпеки, включаючи хешування паролів та генерацію JWT-токенів.

Зовнішній шар системи представлений точками входу, які взаємодіють із зовнішнім світом. Проєкт Api реалізує RESTful інтерфейс із використанням бібліотеки FastEndpoints. Архітектура цього проєкту організована за принципом вертикальних зрізів (Feature Folders), де вся логіка, пов'язана з конкретною функцією (наприклад, Features/Invoices), включаючи ендпоінти та моделі, згрупована разом [24]. Це спрощує навігацію та модифікацію коду. Для виконання фонових задач використовуються безсерверні функції AWS Lambda, реалізовані в проєктах Worker та Emailer. Перший відповідає за ресурсомісткі операції генерації PDF-документів та розрахунок зарплати, а другий — за розсилку повідомлень електронною поштою. Такий розподіл відповідальностей дозволяє ефективно масштабувати окремі компоненти та підвищує стійкість системи до збоїв у фонових процесах.

2.4.2. Функціональна модель та патерни проєктування

Функціональна модель системи базується на подієво-орієнтованій взаємодії компонентів, що дозволяє забезпечити високу масштабованість. Процес обробки даних починається з валідації вхідного запиту на рівні API за допомогою FluentValidation. Після успішної валідації управління передається сервісному шару, який через абстракції репозиторіїв взаємодіє з базою даних. Ключовим моментом є те, що після збереження транзакційних даних система не виконує важкі обчислення синхронно, а публікує подію в чергу повідомлень через ISqsPublisher.

Асинхронна частина системи, реалізована у воркерах, використовує патерн «Стратегія» для вибору відповідного алгоритму обробки повідомлення. Отримавши повідомлення про створення інвойсу, Worker Lambda ініціює процес розрахунку через доменний сервіс EmployeeSalaryCalculator. Цей сервіс агрегує дані про відпрацьовані дні, використовуючи специфікації фільтрації, та виконує бізнес-розрахунки. Результат передається генератору PDF, після чого файл завантажується у хмарне сховище, а по ланцюжку передається подія для сервісу відправки листів.

Для забезпечення гнучкості та супроводжуваності коду в системі широко застосовано патерни проєктування. Патерн «Фабричний метод» (Factory Method) використовується для створення екземплярів обробників повідомлень, що дозволяє легко додавати нові типи подій без модифікації існуючого коду (дотримання принципу Open/Closed) [25]. Патерн «Специфікація» (Specification) у спрощеному вигляді реалізований через динамічну побудову предикатів для вибірки даних [26]. Використання Dependency Injection (впровадження залежностей) пронизує всі шари системи, забезпечуючи слабку зв'язність компонентів та можливість легкого підміну реалізацій, що є критично важливим для модульного тестування [27].

2.5. Формальні алгоритми взаємодії компонентів і обробки даних

Функціонування розробленої системи базується на чітко визначених алгоритмах, які регламентують порядок взаємодії компонентів у розподіленому середовищі та логіку обробки фінансових даних. Формалізація цих процесів

необхідна для верифікації коректності роботи системи та виявлення потенційних колізій при асинхронній обробці запитів.

2.5.1. Алгоритм асинхронної оркестрації процесу білінгу

Основним процедурним компонентом системи є алгоритм наскрізної обробки інвойсу, який охоплює всі шари архітектури — від ініціації запиту користувачем до доставки фінального документу. Враховуючи використання патерну проєктування «Подієво-орієнтована архітектура» (Event-Driven Architecture), алгоритм представляє собою ланцюжок послідовних станів, переходи між якими ініціюються повідомленнями в чергах AWS SQS.

Процес розпочинається з синхронної фази, де API отримує запит на створення інвойсу. На цьому етапі критично важливим є виконання валідації вхідних даних згідно з визначеними контрактами. У разі успішної перевірки система виконує атомарну операцію збереження метаданих інвойсу в базу даних зі статусом «Created». Відразу після цього відбувається публікація події CreatedInvoice у чергу генерації. Цей крок є точкою розгалуження, де процес переходить у асинхронний режим, звільняючи клієнтський потік.

Асинхронна фаза обробки виконується фоновим воркером (Worker Lambda). Алгоритм роботи воркера побудований за принципом ідемпотентності: отримавши повідомлення, він спершу перевіряє актуальність даних у базі [28]. Далі виконується виклик доменного сервісу розрахунків. Результат розрахунків трансформується у PDF-документ, який завантажується у сховище S3. Важливим етапом є оновлення запису в базі даних — додавання посилання на файл, що фактично фіксує успішне завершення етапу генерації. Фінальним кроком є публікація події GeneratedInvoice у чергу відправки, яка активує сервіс розсилки електронної пошти [29]. Така побудова алгоритму гарантує, що навіть у випадку збою на будь-якому етапі, повідомлення повернеться в чергу і буде оброблено повторно, забезпечуючи гарантовану доставку результату. Це дозволяє системі залишатися консистентною та стійкою до тимчасових відмов інфраструктури або повторних обробок повідомлень.

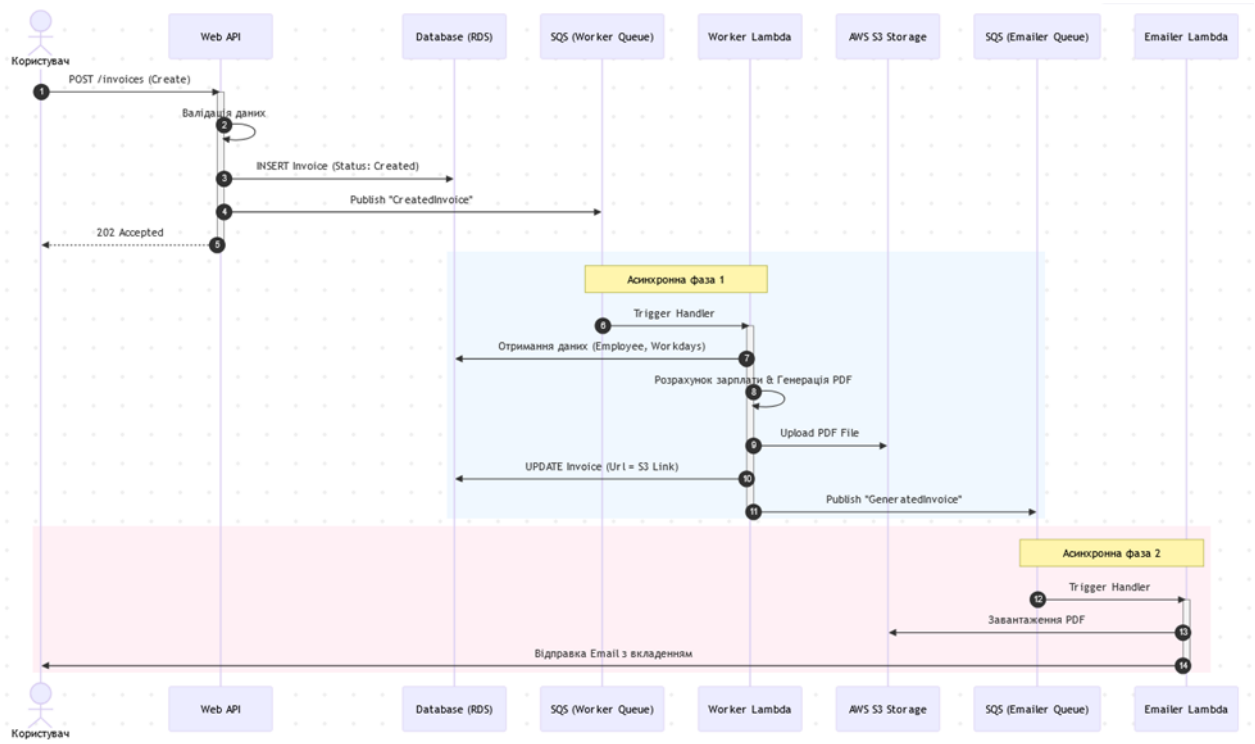


Рис. 2.5. Діаграма послідовності взаємодії компонентів

2.5.2. Алгоритм розрахунку фінансових показників

Ядром бізнес-логіки системи є алгоритм розрахунку суми виплати. Складність даного алгоритму полягає у необхідності динамічного визначення вартості робочої години в залежності від календарних характеристик звітного місяця, що відрізняє його від простих систем з фіксованою ставкою.

Вхідними даними для алгоритму є ідентифікатор співробітника та звітний період (місяць і рік). На першому кроці алгоритм визначає часові межі періоду: перше та останнє число місяця. Наступним кроком виконується ітеративний перебір усіх днів у цьому діапазоні для підрахунку кількості робочих днів (понеділок–п'ятниця), що дозволяє визначити нормативний фонд робочого часу (H_{norm}) при стандартному 8-годинному робочому дні.

Паралельно з цим виконується запит до бази даних для отримання фактично відпрацьованих днів співробітника, які потрапляють у визначений діапазон. На основі отриманих даних обчислюється фактична сума відпрацьованих годин (H_{fact}). Ключовою операцією є розрахунок погодинної ставки (Rate), яка визначається як відношення місячного окладу співробітника до нормативного фонду часу. Фінальна вартість робіт отримується шляхом множення фактичних годин на розраховану ставку

з подальшим математичним округленням до двох знаків після коми [30]. Такий підхід забезпечує справедливий розрахунок оплати праці, враховуючи різну тривалість робочих місяців.

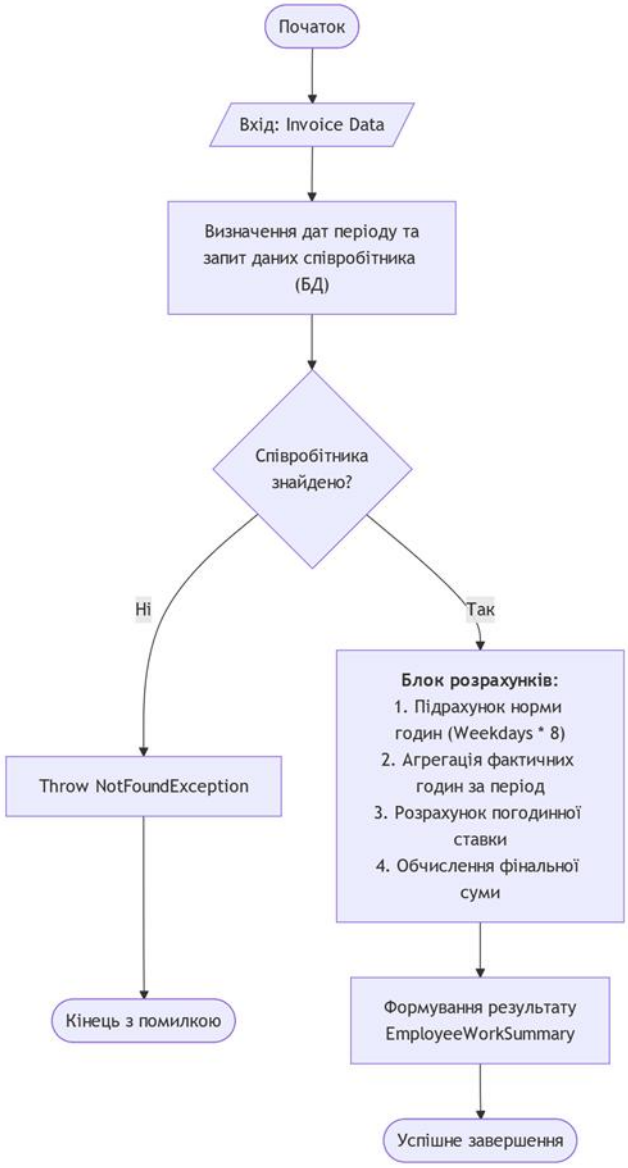


Рис. 2.6. Блок-схема алгоритму розрахунку

2.6. Визначення критеріїв ефективності, надійності та масштабованості системи

Оцінка якості розробленої архітектури вимагає визначення чіткої системи метрик, які дозволяють кількісно та якісно виміряти відповідність системи поставленим вимогам. Враховуючи специфіку безсерверної реалізації (Serverless), традиційні метрики моніторингу, такі як завантаження процесора (CPU Load) або

використання оперативної пам'яті (RAM Usage) сервера, відходять на другий план. Натомість ключовими стають показники, що характеризують поведінку системи при обробці подій, часові затримки та економічну ефективність транзакцій [31].

Для оцінки швидкодії системи пропонується використовувати композитну метрику наскрізної затримки (End-to-End Latency), яка розбивається на дві складові: синхронну та асинхронну.

Першим критичним показником є час відгуку API (T_{api}). Оскільки система використовує патерн асинхронного розвантаження, цей час має бути мінімальним і стабільним, незалежно від складності генерації PDF. Ефективність вважається досягнутою, якщо Tapi не перевищує встановленого порогового значення (наприклад, 500 мс) навіть при високому навантаженні, що свідчить про коректну роботу черги як буфера.

Другим показником є повний час обробки (T_{total}), який визначається як інтервал часу між ініціацією запиту користувачем та отриманням повідомлення про успішне виконання. Для безсерверної архітектури цей показник формулізується наступним чином:

$$T_{total}=T_{api}+T_{queue}+T_{cold}+T_{exec} \quad (2.10)$$

де T_{queue} — час перебування повідомлення в черзі SQS, T_{cold} — час «холодного старту» (Cold Start) Lambda-функції при ініціалізації нового екземпляра контейнера, а T_{exec} — чистий час виконання бізнес-логіки. Мінімізація впливу T_{cold} є специфічним критерієм ефективності для обраного стеку технологій (.NET 8 on AWS Lambda) [32].

Надійність розподіленої системи визначається її здатністю забезпечувати гарантовану доставку повідомлень та цілісність даних в умовах часткових збоїв. Кількісною мірою надійності є коефіцієнт успішних транзакцій (Rsuccess), який розраховується як відношення кількості успішно відправлених інвойсів до загальної кількості запитів.

Важливим аспектом є поведінка системи при виникненні помилок. Ефективність механізму відмовостійкості оцінюється через метрику відновлення

(Recovery Rate) — здатність системи автоматично повторити обробку повідомлення після тимчасового збою (наприклад, тайм-ауту мережі) без втручання адміністратора.

Масштабованість системи оцінюється через її еластичність — здатність динамічно змінювати пропускну здатність (Throughput) відповідно до вхідного навантаження. Критерієм ефективності тут виступає лінійність масштабування: при збільшенні кількості вхідних запитів у N разів, час обробки черги не повинен зростати експоненціально.

Метрикою масштабованості є рівень конкурентності (Concurrency Level) — кількість одночасно запущених екземплярів Worker Lambda, яка повинна автоматично зростати до лімітів, встановлених квотами хмарного провайдера, забезпечуючи паралельну обробку масиву інвойсів [33].

Враховуючи, що економія витрат була однією з ключових причин вибору архітектури, важливим критерієм є вартість однієї транзакції. На відміну від серверних рішень з фіксованою вартістю, для даної системи вартість Cinvoice розраховується як сума мікроплатежів за використані ресурси:

$$C_{invoice} = C_{api} + C_{sqs} + C_{compute}(t) + C_{storage} \quad (2.11)$$

де $C_{compute}$ прямо пропорційна часу виконання функції та виділеному обсягу пам'яті. Критерієм успішності проєкту є значно нижча сукупна вартість володіння (TCO) у порівнянні з розгортанням аналогічного функціоналу на віртуальній машині (EC2) при сценарії періодичного навантаження [34].

2.7. Висновки до розділу

У даному розділі вирішено задачу теоретичного обґрунтування та формального моделювання системи автоматизації фінансових процесів. Перехід від аналізу проблеми до етапу проєктування дозволив сформувати цілісне бачення майбутнього програмного продукту та закласти фундамент для його практичної реалізації.

У ході виконання досліджень отримано наступні результати:

1. Розроблено узагальнену модель системи, яка базується на принципах розподіленої безсерверної архітектури (Serverless). Модель описує систему як сукупність слабозв'язаних компонентів, взаємодія між якими відбувається асинхронно через черги повідомлень. Це дозволило вирішити ключове протиріччя між необхідністю пікової продуктивності та вимогою мінімізації витрат, виявлене у першому розділі.

2. Формалізовано вимоги до програмного забезпечення згідно з міжнародними стандартами IEEE 830-1998 та ISO/IEC 25010. Чітке визначення функціональних вимог (управління сутностями, генерація документів, розсилка) та нефункціональних атрибутів (масштабованість, надійність, безпека) дозволило окреслити межі системи та критерії її валідації. Особливий акцент зроблено на вимогах до відмовостійкості, що є критичним для фінансових систем.

3. Побудовано математичні та логічні моделі бізнес-процесів. Використання апарату теорії множин дозволило формалізувати предметну область та правила валідації даних. Розроблений математичний алгоритм розрахунку вартості робіт, що враховує динаміку робочих днів у календарному місяці, забезпечує точність фінансових нарахувань. Моделювання життєвого циклу інвойсу у вигляді кінцевого автомату дозволило спроєктувати надійний механізм управління станами документу.

4. Спроєктовано архітектуру системи, що поєднує підходи «Чистої архітектури» (Clean Architecture) та вертикальних зрізів (Vertical Slices). Структурна декомпозиція рішення на шари (Domain, Core, Application, Infrastructure, Presentation) забезпечує ізоляцію бізнес-логіки від інфраструктурних деталей, спрощує тестування та супровід коду. Визначено та обґрунтовано використання патернів проєктування: Repository, Specification, Factory Method та Interceptor.

5. Розроблено алгоритми взаємодії компонентів, які регламентують порядок обробки даних у розподіленому середовищі. На відміну від спрощених моделей, запропоновані алгоритми враховують механізми обробки виключних ситуацій

(Exception Handling) та перевірки цілісності даних (Null Checks), що гарантує стабільність роботи фонових процесів та запобігає втраті повідомлень.

6. Визначено критерії ефективності системи. Сформовано систему метрик для оцінки швидкодії (час «холодного старту», наскрізна затримка), надійності (коефіцієнт успішних транзакцій) та економічної доцільності. Це створює базу для проведення експериментальних досліджень.

Таким чином, у даному розділі повністю вирішено задачу формалізації та моделювання системи. Отримані теоретичні результати, моделі та алгоритми є достатнім підґрунтям для переходу до наступного етапу — програмної реалізації прототипу системи та його експериментальної перевірки, що буде розглянуто у третьому розділі.

РОЗДІЛ 3
РЕАЛІЗАЦІЯ ТА ВАЛІДАЦІЯ ПРОТОТИПУ СИСТЕМИ

3.1. Реалізація прототипу системи: компоненти, модулі, потоки даних

3.1.1. Фізична структура проєкту та організація коду

Програмна реалізація системи виконана з використанням екосистеми .NET 8, що забезпечує високу продуктивність, типізовану безпеку та нативну підтримку хмарних середовищ AWS. Рішення (Solution) спроєктовано за принципами Clean Architecture з чітким розділенням на рівні абстракції.

Файлова структура рішення відображає архітектурний поділ на шари: Core (абстракції), Domain (сутності), Application (бізнес-логіка), Persistence (дані) та презентаційні шари (Api, Worker, Emailer). Така організація дозволяє ізолювати залежності та спрощує навігацію по проєкту.

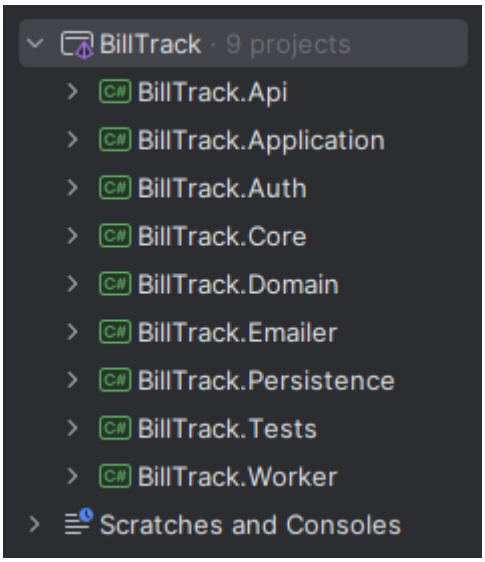


Рис. 3.1. Файлова структура рішення

Ключовою особливістю реалізації API є використання підходу Vertical Slice Architecture в папці Features. Замість групування за типами файлів (Controllers, Models), код згруповано за бізнес-функціями (наприклад, Features/Invoices), що містять ендпоінти, моделі запитів та специфічну логіку поруч.

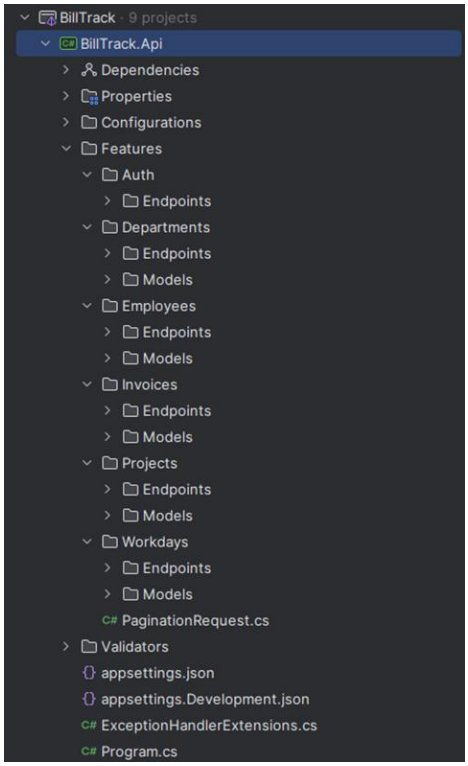


Рис. 3.2. Демонстрація Vertical Slice Architecture в проєкті API

3.1.2. Реалізація інфраструктурного шару та доступу до даних

Персистентність даних забезпечується СУБД PostgreSQL. Для локальної розробки налаштовано контейнеризацію через Docker Compose, що дозволяє розгорнути ідентичне до продуктивного середовище однією командою.

У файлі docker-compose.yml налаштовано прокидання портів 5433:5432, що дозволяє уникнути конфліктів із локальними інсталяціями Postgres на машині розробника (Лістинг 3.1).

Лістинг 3.1

```
services:
  bill_track_db:
    image: postgres:latest
    restart: always
    environment:
      POSTGRES_DB: bill_track_db
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
    ports:
      - '5433:5432'
```

```
    volumes:
      - bill_track_db:/var/lib/postgresql/data
```

На рівні коду взаємодія з базою даних реалізована через ORM Entity Framework Core. Важливим архітектурним елементом є використання Interceptors (Перехоплювачів) для реалізації наскрізного функціоналу аудиту та м'якого видалення. Це дозволяє автоматизувати заповнення службових полів (CreatedAt, UpdatedAt, IsDeleted) без забруднення бізнес-логіки (Лістинг 3.2).

Лістинг 3.2

```
    public static IServiceCollection ConfigureDatabase(this
IServiceCollection services, IConfiguration configuration)
    {
        services.AddDbContext<AppDbContext>((serviceProvider,
options) =>
        {

options.UseNpgsql(configuration.GetConnectionString("DefaultConnection
"))
                .AddInterceptors(

serviceProvider.GetService<SoftDeleteInterceptor>()!,

serviceProvider.GetService<UpdateAuditableEntityInterceptor>()!
                );
        });
        return services;
    }
```

3.1.3. Реалізація REST API та конвеєра обробки запитів

Точка входу API побудована на базі бібліотеки **FastEndpoints**, яка реалізує патерн REPR (Request-Endpoint-Response). Це дозволяє відмовитися від громіздких контролерів.

Конфігурація сервісів винесена у методи розширення (Extension Methods), що забезпечує чистоту файлу Program.cs. Нижче наведено приклад організації конвеєра (Pipeline) з інтеграцією глобальної обробки помилок (Лістинг 3.3).

Лістинг 3.3

```
var builder = WebApplication.CreateBuilder(args);
```

```
builder.Services
    .ConfigureInterceptors()
    .ConfigureAppSettings(builder.Configuration)
    .ConfigureDatabase(builder.Configuration)
    .ConfigureRepositories()
    .ConfigureServices();

builder.Services.AddAWSService<IAmazonSQS>();
builder.Services.AddAWSService<IAmazonS3>();

var app = builder.Build();

await app.UseDatabaseMigrations();
app.UseGlobalExceptionHandler();
app.UseAuthentication();
app.UseAuthorization();
app.UseFastEndpoints(c => { c.Endpoints.RoutePrefix = "api"; });
app.Run();
```

Особливої уваги заслуговує реалізація GlobalExceptionHandler. Він трансформує доменні виключення (наприклад, NotFoundException) у стандартизовані HTTP-відповіді (Problem Details RFC 7807), забезпечуючи консистентність API. Це дозволяє клієнтським застосункам однозначно обробляти помилки та спрощує діагностику проблем у продуктивному середовищі.

Лістинг 3.4

```
var (statusCode, errorMessage) = exception switch
{
    NotFoundException notFoundException =>
(StatusCodes.Status404NotFound, notFoundException.Message),
    UnauthorizedAccessException unauthorizedException =>
(StatusCodes.Status401Unauthorized, unauthorizedException.Message),
    _ => (StatusCodes.Status500InternalServerError, "An
unexpected error occurred.")
};

logger.LogError(exception, "[{@exceptionType}] at [{@route}] due
to [{@reason}]",
    exceptionType, route, reason);

await ctx.Response.WriteAsJsonAsync(new InternalErrorResponse
{
    Status = "Error",
    Code = statusCode,
    Reason = errorMessage
});
```

3.1.4. Реалізація бізнес-логіки та динамічної фільтрації

Шар бізнес-логіки (Application) містить сервіси, які оркеструють роботу з репозиторіями та чергами. Для реалізації гнучких пошукових запитів розроблено механізм динамічної побудови предикатів на основі дерев виразів (Expression Trees).

Клас PredicateBuilder дозволяє динамічно комбінувати умови фільтрації (AND/OR), а EntityFilter надає зручний інтерфейс для побудови специфікацій (Лістинг 3.5).

Лістинг 3.5

```
      public Task<PagedResult<Workday>> GetAllWorkdaysPagedAsync(
              int pageNumber,
              int pageSize,
              string? sortByDate,
              DateOnly? filterByDate,
              Guid? filterByEmployee)
          {
              var filter =
EntityFilter.BuildWorkdayFilter(filterByDate, filterByEmployee);
              var orderBy = CreateOrderByFunc<Workday,
DateOnly>(sortByDate, w => w.Date);

              Expression<Func<Workday, object>>[] includes = { w =>
w.Employee };

              return GetAllPagedAsync(pageNumber, pageSize, filter,
orderBy, includes);
          }

      public static Expression<Func<Workday, bool>>
BuildWorkdayFilter(DateOnly? date, Guid? employeeId)
      {
          return BuildFilter<Workday>(
              (date.HasValue, w => w.Date == date.Value),
              (employeeId.HasValue, w => w.EmployeeId ==
employeeId.Value)
          );
      }

      private static Expression<Func<T, bool>> BuildFilter<T>(params
(bool condition, Expression<Func<T, bool>> predicate)[] filters)
      {
          Expression<Func<T, bool>>? combinedFilter = null;
          foreach (var (condition, predicate) in filters)
          {
```

```
            if (condition)
                combinedFilter = combinedFilter == null ?
predicate : combinedFilter.And(predicate);
        }
        return combinedFilter ?? PredicateBuilder.True<T>();
    }
```

Такий підхід дозволяє уникнути написання десятків окремих методів у репозиторії для кожної комбінації параметрів пошуку.

3.1.5. Реалізація асинхронних воркерів (AWS Lambda)

Фонова обробка винесена у безсерверні функції. Специфіка .NET на AWS Lambda полягає у відсутності стандартного Startup.cs, тому конфігурація Dependency Injection виконується вручну у конструкторі класу Function (Лістинг 3.6).

Лістинг 3.6

```
    public Function()
    {
        var serviceCollection = new ServiceCollection();
        ConfigureServices(serviceCollection);
        var serviceProvider =
serviceCollection.BuildServiceProvider();

        _messageProcessor =
serviceProvider.GetRequiredService<IMessageProcessor>();
    }

    private void ConfigureServices(IServiceCollection services)
    {
        var configuration = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile($"appsettings.json")
            .AddEnvironmentVariables()
            .Build();

        services.AddAWSService<IAmazonS3>();
        services.AddAWSService<IAmazonSQS>();

        services.ConfigureRepositories().ConfigureServices();
    }
```

Логіка обробки повідомлення про створення інвойсу реалізована в CreatedInvoiceMessageHandler. Вона включає валідацію, генерацію PDF, завантаження

в S3 та відправку повідомлення далі по ланцюжку. Реалізовано структуроване логування через LogContext для трасування транзакцій (Лістинг 3.7).

Лістинг 3.7

```
      public async Task HandleMessageAsync(CreatedInvoice
invoiceMessage)
      {
          using (LogContext.PushProperty("InvoiceId",
invoiceMessage.InvoiceId))
          {
              var fileName = $"invoice-
{invoiceMessage.InvoiceId}.pdf";
              try
              {
                  var invoice = await
GetInvoiceWithEmployee(invoiceMessage.InvoiceId);
                  if (invoice == null) throw new
NotFoundException(invoiceMessage.InvoiceId);

                  await GenerateAndUploadPdf(invoiceMessage.InvoiceId,
fileName);

                  var invoiceUrl = GenerateInvoiceUrl(fileName);
                  await UpdateInvoiceUrl(invoice, invoiceUrl);

                  var generatedMessage =
CreateGeneratedInvoiceMessage(invoice, fileName, ...);
                  await
_sqsPublisher.PublishMessageAsync(_awsConfiguration.Value.EmailQueue,
generatedMessage);
              }
              catch (Exception ex)
              {
                  Log.Logger.Error(ex, "Failed to process invoice");
                  throw;
              }
          }
      }
```

3.1.6. Реалізація сервісу нотифікацій (Emailer Lambda)

Сервіс відправки листів реалізований як окрема Lambda-функція. Для роботи з SMTP використовується бібліотека FluentEmail. Конфігурація SMTP-клієнта завантажується з налаштувань EmailSettings, що дозволяє гнучко змінювати поштового провайдера без перекомпіляції коду (Лістинг 3.8).

Лістинг 3.8

```
var smtp = new SmtpClient
{
    Host = emailSettings.EmailHost,
    Port = emailSettings.EmailSmtpPort,
    EnableSsl = true,
    Credentials = new
NetworkCredential(emailSettings.SenderEmail,
emailSettings.SenderPassword)
};
Email.DefaultSender = new SmtpSender(smtp);

var emailResponse = await Email
    .From(_emailSettings.SenderEmail)
    .To(invoice.EmailTo)
    .Subject($"Invoice for {DateTime.Now.ToShortDateString()}")
    .Body(emailBody)
    .Attach(new Attachment
    {
        Data = invoiceFileStream,
        ContentType = "application/pdf",
        Filename = invoice.FileName
    })
    .SendAsync();
```

Таким чином, програмна реалізація повністю відповідає спроєктованій архітектурі, забезпечуючи надійну асинхронну обробку даних, модульність та масштабованість.

3.2. Інтеграція та налаштування середовища виконання (Cloud Deployment)

Для розгортання прототипу системи у хмарне середовище AWS було обрано стратегію ручного розгортання (Manual Deployment) з використанням інтерфейсу командного рядка (CLI). Такий підхід забезпечує максимальний контроль над процесом оновлення функцій на етапі розробки та налагодження, дозволяючи розробнику миттєво бачити результати компіляції та помилки розгортання в терміналі. Водночас ручне розгортання дозволяє поступово інтегрувати нові компоненти без ризику порушення роботи вже існуючих сервісів. Крім того, такий підхід сприяє глибшому розумінню роботи інфраструктури та механізмів взаємодії безсерверних функцій у хмарному середовищі.

3.2.1. Підготовка інструментарію розгортання

Для інтеграції .NET-застосунків з сервісами AWS використано розширення AWS Lambda Tools for .NET. Це глобальний інструмент, який дозволяє керувати життєвим циклом безсерверних функцій безпосередньо з командного рядка [35].

Встановлення інструменту виконано командою:

```
dotnet tool install -g Amazon.Lambda.Tools
```

Автентифікація локального середовища в хмарі AWS налаштована через AWS CLI. У системі створено профіль розробника з правами адміністратора, конфігурація якого зберігається локально. Це дозволяє інструментам розгортання автоматично підписувати запити до API AWS без необхідності передачі ключів у кожній команді.

3.2.2. Конфігурація параметрів деплою

Щоб уникнути ручного введення довгих параметрів (ARN ролей, розмір пам'яті, тайм-аути) при кожному розгортанні, конфігурація для кожної функції винесена у файл aws-lambda-tools-defaults.json.

Наприклад, для сервісу розсилки (Emailer) конфігурація має вигляд (Лістинг 3.9).

Лістинг 3.9

```
{
  "profile": "default",
  "region": "eu-central-1",
  "configuration": "Release",
  "function-architecture": "x86_64",
  "function-runtime": "dotnet8",
  "function-memory-size": 512,
  "function-timeout": 30,
  "function-handler": "BillTrack.Api",
  "framework": "net8.0",
  "function-name": "BillTrackApi",
  "package-type": "Zip",
  "function-role":
"arn:aws:iam::YOUR_ACCOUNT_ID:role/YOUR_LAMBDA_ROLE"
}
```

Цей файл пов'язує локальну збірку з конкретною Lambda-функцією у хмарі та визначає її ресурси.

3.2.3. Процес розгортання компонентів

Безпосереднє розгортання виконується через термінал IDE JetBrains Rider. Процес складається з компіляції коду в режимі Release, пакування бінарних файлів у ZIP-архів та завантаження його в AWS Lambda [36].

Команда для оновлення коду Worker-сервісу:

```
cd BillTrack.Api
dotnet lambda deploy-function BillTrackApi
```

Під час виконання команди інструмент автоматично:

1. Відновлює NuGet-пакети (dotnet restore).
2. Компілює проєкт (dotnet publish).
3. Оновлює код існуючої функції в AWS або створює нову, якщо вона відсутня.

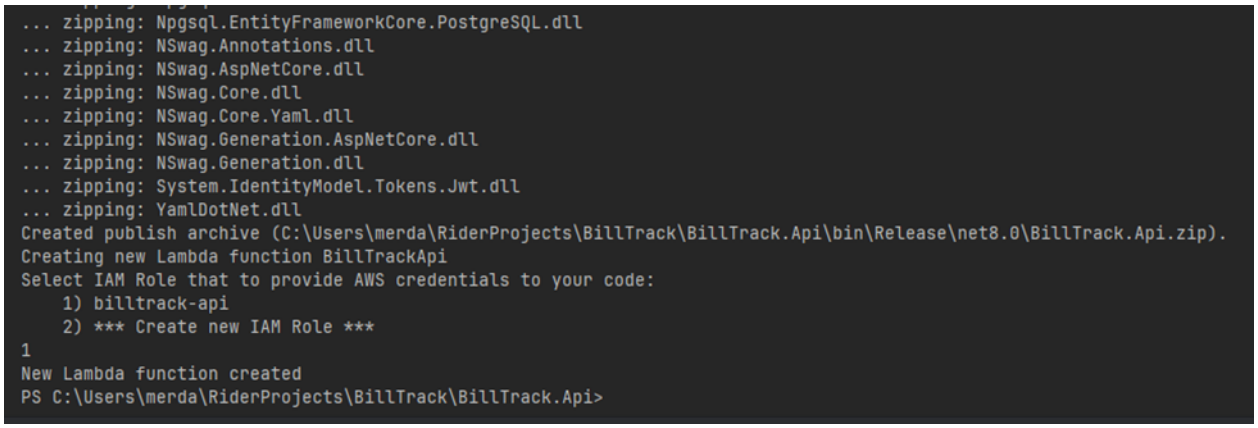


Рис. 3.3. Лог успішного розгортання функції через термінал

Після завершення роботи інструменту командного рядка (CLI) та отримання повідомлення про успішне завантаження артефактів, критично важливим етапом є верифікація стану розгорнутих ресурсів безпосередньо у хмарному середовищі.

Перевірка проводиться через вебконсоль AWS Management Console, яка надає повний візуальний контроль над інфраструктурою.

У розділі Lambda -> Functions необхідно пересвідчитися у наявності двох ключових функцій: BillTrackWorker (відповідає за бізнес-логіку) та BillTrackEmailer (відповідає за комунікацію). Окрім самого факту наявності функцій, перевірці підлягають їх конфігураційні параметри:

- Обсяг виділеної пам'яті: Перевірка відповідності значенню, заданому в aws-lambda-tools-defaults.json (512 МБ). Це важливо, оскільки в моделі AWS Lambda обчислювальна потужність процесора (vCPU) виділяється пропорційно до обсягу пам'яті.
- Тайм-аути виконання: Перевірка встановленого ліміту часу (30 секунд), що запобігає «зависанню» функцій та надмірним фінансовим витратам у разі помилок у коді.
- IAM Ролі: Перевірка коректності призначення ролей доступу, які дозволяють функціям взаємодіяти з іншими сервісами (читати з SQS, писати в S3).

Окремим і надзвичайно важливим етапом налаштування є конфігурація змінних середовища (Environment Variables). Оскільки зберігання конфіденційних даних (connection strings, JWT secret keys, SMTP credentials) у репозиторії коду є грубим порушенням політик безпеки, ці дані вводяться вручну через захищений інтерфейс консолі AWS. Змінні шифруються на стороні AWS за допомогою сервісу KMS (Key Management Service) і розшифровуються автоматично лише в момент запуску контейнера функції [37].

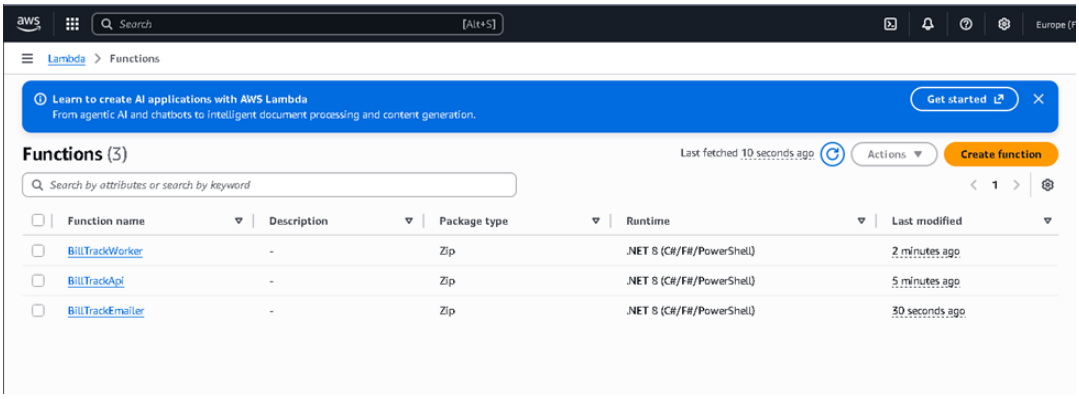


Рис. 3.4. Список розгорнутих мікросервісів у консолі AWS

Таким чином, налаштоване середовище дозволяє виконувати швидкі ітерації розробки, доставляючи зміни коду в хмару за 30-40 секунд, що значно пришвидшує процес налагодження та тестування порівняно з класичним деплоєм на віртуальні машини.

3.3. Експериментальна перевірка алгоритмів та процесів автоматизації

Метою експериментальної перевірки є комплексна верифікація коректності взаємодії компонентів розподіленої системи в реальних умовах експлуатації. Тестування проводилося за сценарієм «End-to-End» (наскрізний процес), що охоплює повний життєвий цикл генерації інвойсу: від ініціації HTTP-запиту клієнтом до отримання кінцевого результату у вигляді електронного листа. Такий підхід дозволяє перевірити не лише роботу окремих модулів (Unit testing), але й коректність інтеграційних зв'язків між хмарними сервісами.

3.3.1. Ініціація процесу через REST API

На першому етапі перевірялася робота точки входу API та валідаторів. За допомогою інструменту Postman, який є стандартом де-факто для тестування REST API, було сформовано HTTP POST-запит на ендпоінт /api/invoices/create [38]. Цей запит імітує дію користувача (адміністратора або бухгалтера) у вебінтерфейсі системи.

У тілі запиту у форматі JSON передано дані для генерації звіту за грудень 2025 року для тестового співробітника. Структура запиту повністю відповідає контракту даних DTO (Data Transfer Object), визначеному в шарі Core, що гарантує правильну десеріалізацію та обробку інформації. Це дозволяє системі коректно інтерпретувати вхідні дані та забезпечує сумісність між різними компонентами архітектури.

```
{
  "month": 12,
  "year": 2025,
  "employeeId": "10671fd4-ed4b-40fd-9d72-28a7e7ba3b32"
}
```

Як видно з рис. 3.5, сервер повернув статус код 200 OK (або, що більш архітектурно правильно для асинхронних систем — 202 Accepted). Цей код свідчить про те, що запит успішно пройшов первинну перевірку, був прийнятий сервером, а задача на генерацію документа поставлена в чергу обробки. Важливо зазначити, що час відповіді був мінімальним, оскільки сервер не виконував важку операцію генерації PDF у тому ж потоці.

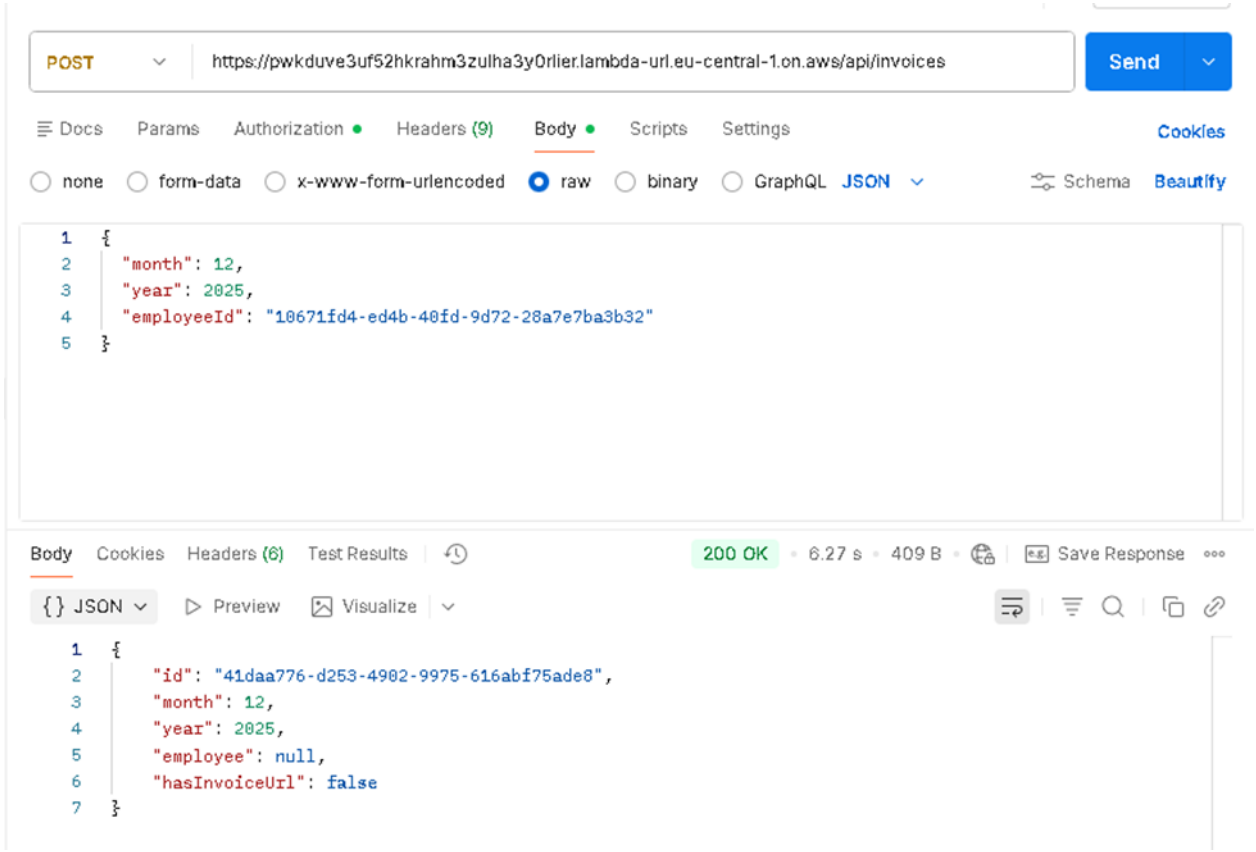


Рис. 3.5. Ініціація створення інвойсу через Postman

Для перевірки надійності системи та коректності роботи механізму валідації (FluentValidation) було проведено серію ретельних негативних тестів. Зокрема, для одного із сценаріїв було надіслано запит із некоректними бізнес-даними, наприклад, вказано неіснуючий 13-й місяць, що виходить за межі допустимого діапазону. Система коректно перехопила некоректні дані ще до етапу обробки, відхилила запит із HTTP-кодом 400 Bad Request та повернула детальний опис помилки у тілі відповіді (Problem Details), як показано на рис. 3.6. Це наочно підтверджує ефективність захисту системи від введення некоректних даних і гарантує стабільну роботу сервісу.

Аналогічні тести проводилися й для інших типів помилок, зокрема відсутніх обов'язкових полів, некоректних форматів значень або невідповідностей типів даних. У всіх випадках поведінка системи була передбачуваною та повністю відповідала специфікації API, що забезпечує узгодженість та передбачуваність роботи. Такий підхід значно спрощує діагностику помилок для клієнтських застосунків та підвищує зручність їх інтеграції. Крім того, рання валідація зменшує навантаження на бізнес-логіку та фонові сервіси, дозволяючи ефективніше використовувати ресурси системи та підвищувати загальну продуктивність.

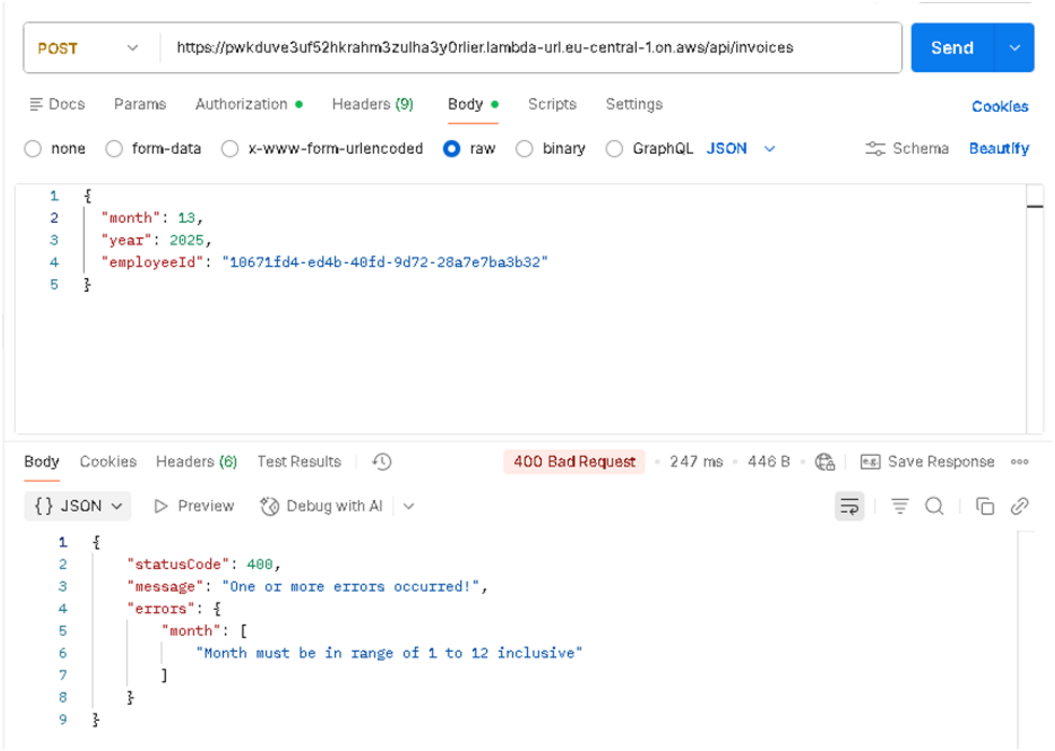
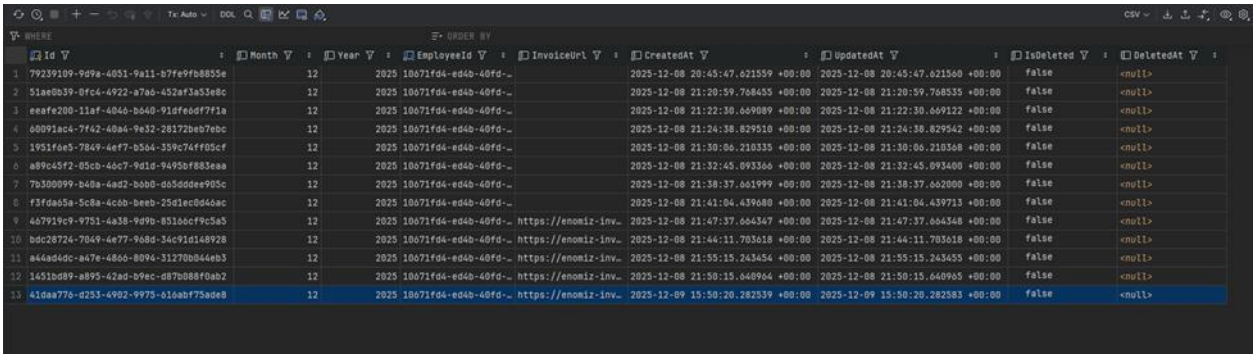


Рис. 3.6. Робота механізму валідації вхідних даних

3.3.2. Верифікація асинхронної обробки (Logs & Database)

Після успішного прийняття запиту API, фокус перевірки переміщується на шар даних та моніторингу. Для підтвердження того, що процес дійсно розпочався, було виконано SQL-запит до таблиці Invoices у базі даних PostgreSQL. Результат вибірки підтвердив створення нового запису з відповідним EmployeeId та звітним періодом.

Особливістю асинхронної обробки є те, що дані оновлюються поступово (Eventual Consistency). На рис. 3.7 відображено стан запису вже після завершення обробки воркером: поле InvoiceUrl, яке спочатку було пустим (NULL), тепер містить посилання на згенерований файл у хмарному сховищі. Це є прямим доказом успішного завершення роботи бізнес-логіки. Подібна стратегія забезпечує баланс між продуктивністю та надійністю системи, дозволяючи обробляти великі обсяги даних без блокувань. Крім того, вона дає можливість поступово відображати актуальний стан інформації для користувачів і зовнішніх сервісів.



	Id	Month	Year	EmployeeId	InvoiceUrl	CreatedAt	UpdatedAt	IsDeleted	DeletedAt
1	79239109-9d9a-4051-9a11-b7fe9fb8855e	12	2025	18671fd4-ed4b-40fd-…		2025-12-08 20:45:47.621559 +00:00	2025-12-08 20:45:47.621560 +00:00	false	<null>
2	51ae0b39-0fc4-4922-a7a6-452af3a53e8c	12	2025	18671fd4-ed4b-40fd-…		2025-12-08 21:20:59.768455 +00:00	2025-12-08 21:20:59.768535 +00:00	false	<null>
3	eeafe200-11af-4040-b640-91df60df7f1a	12	2025	18671fd4-ed4b-40fd-…		2025-12-08 21:22:30.669089 +00:00	2025-12-08 21:22:30.669122 +00:00	false	<null>
4	60091ac4-7f42-40a4-9e32-28172beb7ebc	12	2025	18671fd4-ed4b-40fd-…		2025-12-08 21:24:38.829510 +00:00	2025-12-08 21:24:38.829542 +00:00	false	<null>
5	1951f6e5-7849-4ef7-b5b4-359c74ff05cf	12	2025	18671fd4-ed4b-40fd-…		2025-12-08 21:30:06.210335 +00:00	2025-12-08 21:30:06.210368 +00:00	false	<null>
6	a89c45f2-05cb-46c7-9d1d-9495bf883eaa	12	2025	18671fd4-ed4b-40fd-…		2025-12-08 21:32:45.093366 +00:00	2025-12-08 21:32:45.093400 +00:00	false	<null>
7	7b300099-b40a-4ad2-b0b0-d65dddee905c	12	2025	18671fd4-ed4b-40fd-…		2025-12-08 21:38:37.661999 +00:00	2025-12-08 21:38:37.662000 +00:00	false	<null>
8	f3fda65a-5c8a-4c6b-beeb-25d1ec8d46ac	12	2025	18671fd4-ed4b-40fd-…		2025-12-08 21:41:04.439680 +00:00	2025-12-08 21:41:04.439713 +00:00	false	<null>
9	467919c9-9751-4a38-9d9b-851b6cf9c5a5	12	2025	18671fd4-ed4b-40fd-…	https://enomiz-inv…	2025-12-08 21:47:37.664347 +00:00	2025-12-08 21:47:37.664348 +00:00	false	<null>
10	bdc28724-7849-4e77-968d-34c91d148928	12	2025	18671fd4-ed4b-40fd-…	https://enomiz-inv…	2025-12-08 21:44:11.703618 +00:00	2025-12-08 21:44:11.703618 +00:00	false	<null>
11	a44ad4dc-a47e-4866-8094-31270b844eb3	12	2025	18671fd4-ed4b-40fd-…	https://enomiz-inv…	2025-12-08 21:55:15.243454 +00:00	2025-12-08 21:55:15.243455 +00:00	false	<null>
12	1451bd89-a895-42ad-b9ec-d87b088f0ab2	12	2025	18671fd4-ed4b-40fd-…	https://enomiz-inv…	2025-12-08 21:50:15.640964 +00:00	2025-12-08 21:50:15.640965 +00:00	false	<null>
13	41daa776-d253-4902-9975-b16abf75ade8	12	2025	18671fd4-ed4b-40fd-…	https://enomiz-inv…	2025-12-09 15:50:20.282539 +00:00	2025-12-09 15:50:20.282583 +00:00	false	<null>

Рис. 3.7. Стан запису в базі даних після обробки воркером

Далі проводиться детальний аналіз логів безсерверної функції BillTrackWorker за допомогою сервісу AWS CloudWatch, що дозволяє відстежувати послідовність виконання операцій та виявляти потенційні проблеми у роботі сервісу. Лог-стрім, представлений на рис. 3.8, демонструє етапи обробки повідомлень. Завдяки такій системі логування можна відслідковувати як успішні виконання, так і виняткові ситуації, що значно спрощує діагностику та підтримку функції.

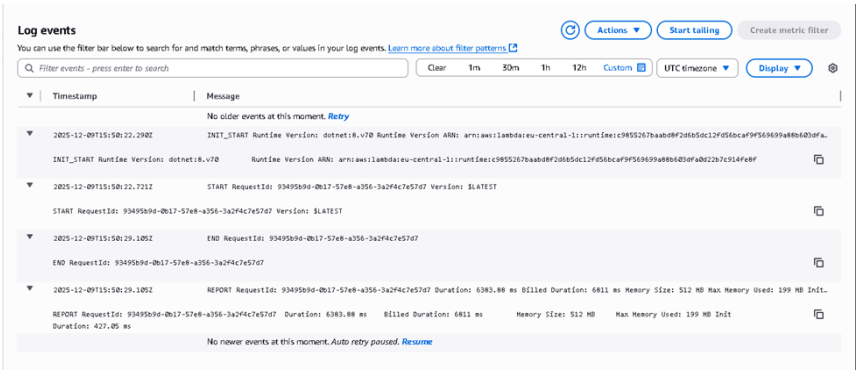


Рис. 3.8. Логи виконання Worker Lambda в CloudWatch

3.3.3. Перевірка генерації артефактів (S3 Storage)

Результатом роботи воркера є фізичний файл — цифровий документ. Для верифікації його наявності було здійснено перегляд вмісту бакету (Bucket) в сервісі Amazon S3 через консоль управління. Перевірка підтверджує появу нового об'єкта з розширенням .pdf.

Ім'я файлу сформовано згідно з патерном invoice-{guid}.pdf, що забезпечує унікальність імен і запобігає колізіям при масовій генерації документів. Важливо, що файл зберігається у приватному бакеті, що унеможливлює несанкціонований доступ до нього з інтернету без спеціального підписаного посилання.

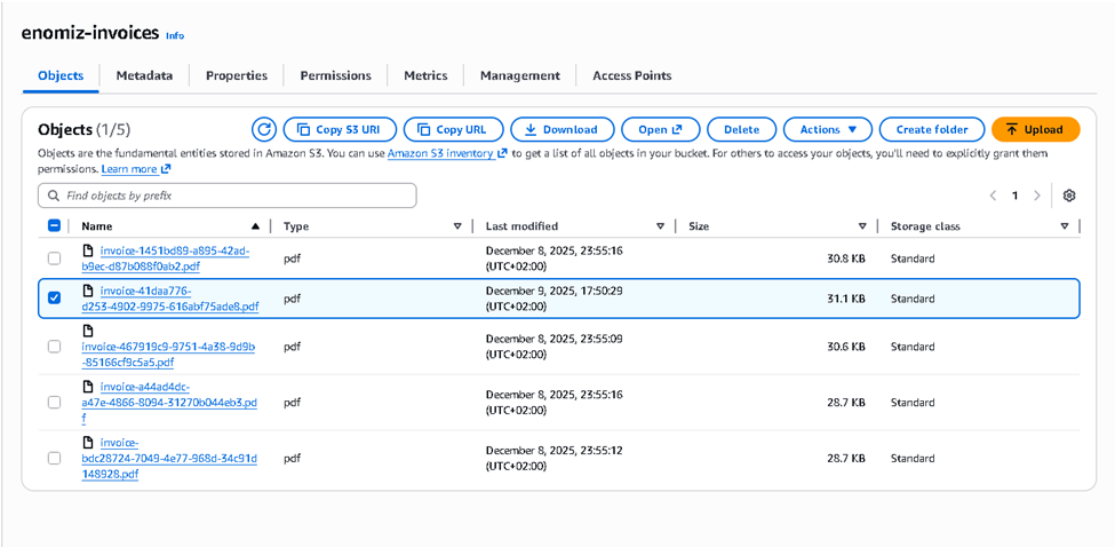


Рис. 3.9. Згенерований файл у хмарному сховищі S3

3.3.4. Отримання результату (Email Delivery)

Фінальним етапом наскрізного тестування є перевірка доставки повідомлення кінцевому користувачу. Сервіс BillTrackEmailer успішно обробив подію GeneratedInvoice з черги розсилки, завантажив файл з S3 та ініціював відправку листа через SMTP-сервер.

На рис. 3.10 продемонстровано скріншот поштової скриньки отримувача. Лист надійшов успішно, містить коректну тему, персоналізоване звернення у тілі повідомлення та, що найголовніше, прикріплений PDF-файл. Розмір вкладення

відповідає розміру файлу в S3, що свідчить про цілісність передачі даних. Фінальним етапом наскрізного тестування є перевірка доставки повідомлення кінцевому користувачу. Сервіс BillTrackEmailer успішно обробив подію GeneratedInvoice з черги розсилки, завантажив файл з S3 та ініціював відправку листа через SMTP-сервер. Лист надійшов успішно, містить коректну тему, персоналізоване звернення у тілі повідомлення та, що найголовніше, прикріплений PDF-файл. Розмір вкладення відповідає розміру файлу в S3, що свідчить про цілісність передачі даних.

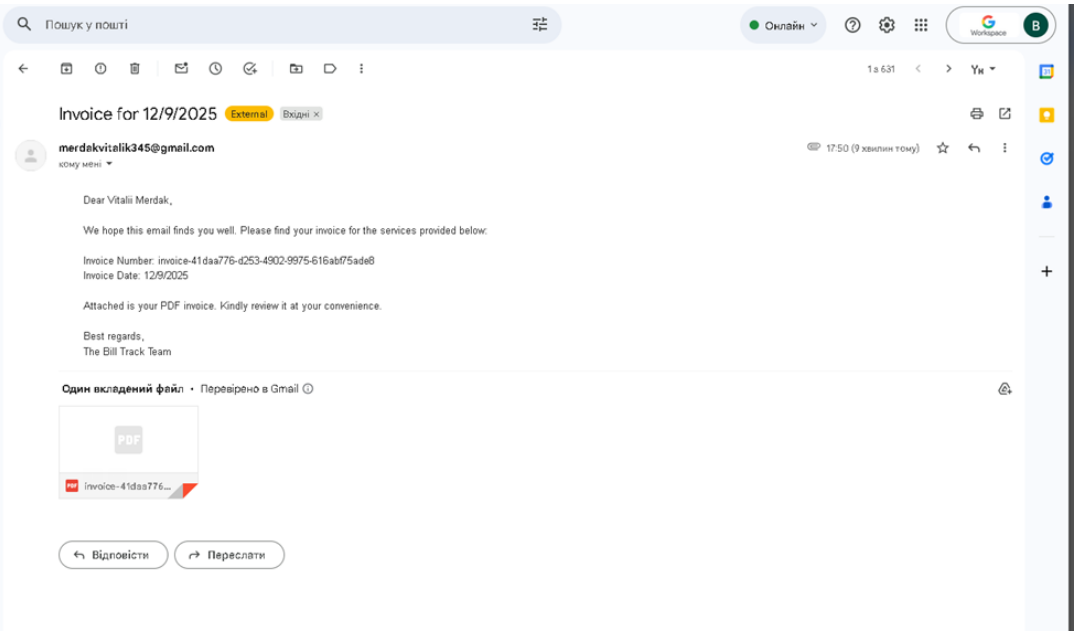
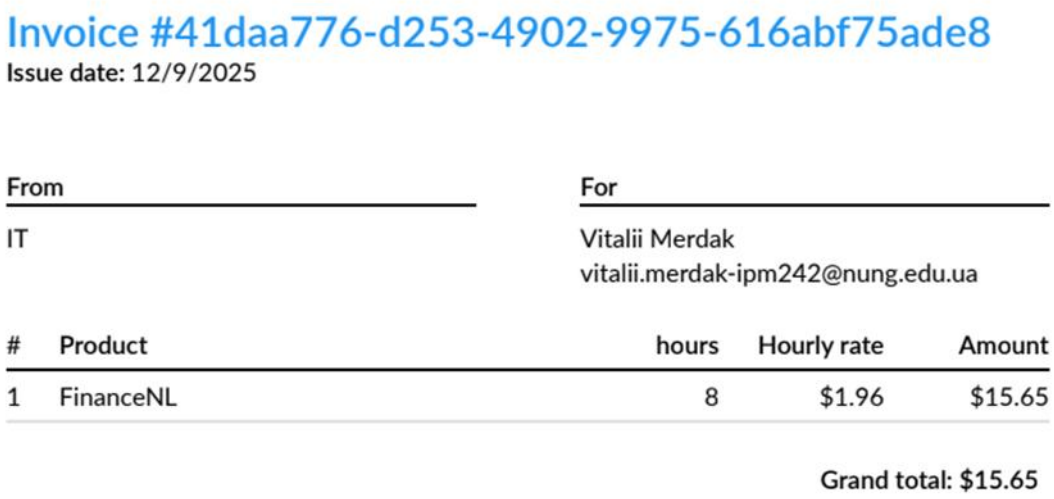


Рис. 3.10. Отриманий електронний лист із інвойсом

Відкриття вкладеного PDF-файлу підтверджує коректність роботи бібліотеки QuestPDF та логіки рендерингу [39]. Документ має професійне оформлення, містить таблицю з деталізацією розрахунків, коректні дані співробітника та математично вірну підсумкову суму. Це остаточно підтверджує, що вся логіка системи — від бази даних до генерації візуального представлення — працює коректно.

Таким чином, експериментальна перевірка підтвердила повну працездатність та коректну інтеграцію усього ланцюжка компонентів: API → DB → SQS → Worker → S3 → Emailer. Кожен етап обробки даних працює послідовно та надійно, що забезпечує стабільну генерацію, збереження та доставку фінансових документів без втрат або збоїв.



Invoice #41daa776-d253-4902-9975-616abf75ade8
Issue date: 12/9/2025

From	For
IT	Vitalii Merdak vitalii.merdak-ipm242@nung.edu.ua

#	Product	hours	Hourly rate	Amount
1	FinanceNL	8	$1.96	$15.65

Grand total: $15.65

Рис. 3.11. Візуалізація згенерованого PDF-документу

3.4. Тестування продуктивності, надійності та масштабованості

Етап тестування системи включав перевірку функціональної коректності компонентів через модульне тестування (unit testing), а також оцінку продуктивності, надійності та масштабованості через навантажувальне тестування. Комплексний підхід до тестування дозволив забезпечити високу якість коду та підтвердити відповідність системи нефункціональним вимогам.

3.4.1. Модульне тестування (Unit Testing)

Модульне тестування є критично важливим етапом розробки, що дозволяє перевірити коректність роботи окремих компонентів системи в ізоляції. Для реалізації тестів використовувався фреймворк xUnit в поєднанні з бібліотекою Moq для створення mock-об'єктів та Bogus для генерації тестових даних.

Тестовий проєкт Tests організований за принципом дзеркального відображення структури основного проєкту, що забезпечує легку навігацію та

підтримку. Всі тести дотримуються паттерну AAA (Arrange-Act-Assert), що робить їх читабельними та зрозумілими.

Сервіс автентифікації є критично важливим компонентом системи, відповідальним за перевірку облікових даних користувачів та генерацію JWT-токенів. Тестування цього сервісу включає перевірку як успішних сценаріїв, так і обробку помилкових ситуацій (Лістинг 3.10).

Лістинг 3.10.

```
using System.Linq.Expressions;
using BillTrack.Auth.Jwt;
using BillTrack.Core.Exceptions;
using BillTrack.Core.Interfaces.Repositories;
using BillTrack.Core.Interfaces.Services;
using BillTrack.Core.Interfaces.Utils;
using BillTrack.Domain.Entities;
using Microsoft.Extensions.Configuration;
using Moq;

namespace BillTrack.Tests.Auth.Jwt;

public class AuthServiceTests
{
    private readonly Mock<IGenericRepository<User>>
_mockUserRepository;
    private readonly Mock<IConfiguration> _mockConfiguration;
    private readonly Mock<IPasswordHasher> _mockPasswordHasher;
    private readonly Mock<IJwtTokenCreator>
_mockJwtTokenCreator;
    private readonly AuthService _authService;

    public AuthServiceTests()
    {
        _mockUserRepository = new
Mock<IGenericRepository<User>>();
        _mockConfiguration = new Mock<IConfiguration>();
        _mockPasswordHasher = new Mock<IPasswordHasher>();
        _mockJwtTokenCreator = new Mock<IJwtTokenCreator>();

        _authService = new AuthService(
            _mockUserRepository.Object,
            _mockConfiguration.Object,
            _mockPasswordHasher.Object,
            _mockJwtTokenCreator.Object
        );
    }

    [Fact]
```

```
        public async Task
GenerateToken_ValidCredentials_ReturnsToken()
        {
            // Arrange
            var username = "test@example.com";
            var password = "password123";
            var userId = Guid.NewGuid();
            var user = new User { Id = userId, Email = username,
Password = "hashedPassword" };

            _mockUserRepository.Setup(r =>
r.FindAsync(It.IsAny<Expression<Func<User, bool>>>()))
                .ReturnsAsync(user);
            _mockPasswordHasher.Setup(h =>
h.Verify(It.IsAny<string>(), password))
                .Returns(true);
            _mockConfiguration.Setup(c => c["JwtSecretKey"])
                .Returns("ThisIsAVeryLongSecretKeyForTesting");
            _mockJwtTokenCreator.Setup(j =>
j.CreateToken(It.IsAny<Guid>(), It.IsAny<string>()))
                .Returns("mocked_jwt_token");

            // Act
            var token = await _authService.GenerateToken(username,
password);

            // Assert
            Assert.NotNull(token);
            Assert.NotEmpty(token);
        }

        [Fact]
        public async Task
GenerateToken_InvalidUsername_ThrowsNotFoundException()
        {
            // Arrange
            var username = "nonexistent@example.com";
            var password = "password123";

            _mockUserRepository.Setup(r =>
r.FindAsync(It.IsAny<Expression<Func<User, bool>>>()))
                .ReturnsAsync((User)null);
            _mockJwtTokenCreator.Setup(j =>
j.CreateToken(It.IsAny<Guid>(), It.IsAny<string>()))
                .Returns("mocked_jwt_token");

            // Act & Assert
            await Assert.ThrowsAsync<NotFoundException>(() =>
                _authService.GenerateToken(username, password));
        }

        [Fact]
```

```
        public async Task
GenerateToken_InvalidPassword_ThrowsNotFoundException()
        {
            // Arrange
            var username = "test@example.com";
            var password = "wrongPassword";
            var user = new User { Id = Guid.NewGuid(), Email =
username, Password = "hashedPassword" };

            _mockUserRepository.Setup(r =>
r.FindAsync(It.IsAny<Expression<Func<User, bool>>>()))
                .ReturnsAsync(user);
            _mockPasswordHasher.Setup(h =>
h.Verify(It.IsAny<string>(), password))
                .Returns(false);
            _mockJwtTokenCreator.Setup(j =>
j.CreateToken(It.IsAny<Guid>(), It.IsAny<string>()))
                .Returns("mocked_jwt_token");

            // Act & Assert
            await Assert.ThrowsAsync<NotFoundException>(() =>
                _authService.GenerateToken(username, password));
        }

        [Fact]
        public async Task
GenerateToken_MissingJwtSecretKey_ThrowsInvalidOperationException()
        {
            // Arrange
            var username = "test@example.com";
            var password = "password123";
            var user = new User { Id = Guid.NewGuid(), Email =
username, Password = "hashedPassword" };

            _mockUserRepository.Setup(r =>
r.FindAsync(It.IsAny<Expression<Func<User, bool>>>()))
                .ReturnsAsync(user);
            _mockPasswordHasher.Setup(h =>
h.Verify(It.IsAny<string>(), password))
                .Returns(true);
            _mockConfiguration.Setup(c => c["JwtSecretKey"])
                .Returns((string)null);
            _mockJwtTokenCreator.Setup(j =>
j.CreateToken(It.IsAny<Guid>(), It.IsAny<string>()))
                .Returns("mocked_jwt_token");

            // Act & Assert
            await Assert.ThrowsAsync<InvalidOperationException>(()
=>
                _authService.GenerateToken(username, password));
        }
    }
```

Тести для AuthService охоплюють чотири ключові сценарії: успішну генерацію токену при валідних облікових даних, обробку неіснуючого користувача, перевірку невірного пароля та відсутність конфігураційного параметру JWT секретного ключа. Використання Moq дозволяє ізолювати тестований сервіс від зовнішніх залежностей, таких як база даних та конфігурація.

Компонент PasswordHasher відповідає за безпечне зберігання паролів користувачів через криптографічне хешування. Тестування цього компоненту критично важливе для забезпечення безпеки системи (Лістинг 3.11).

Лістинг 3.11

```
using BillTrack.Auth.Utils;

namespace BillTrack.Tests.Auth.Utils;

public class PasswordHasherTests
{
    private readonly PasswordHasher _passwordHasher;

    public PasswordHasherTests()
    {
        _passwordHasher = new PasswordHasher();
    }

    [Fact]
    public void Hash_ShouldReturnNonEmptyString()
    {
        // Arrange
        var password = "TestPassword123";

        // Act
        var hashedPassword = _passwordHasher.Hash(password);

        // Assert
        Assert.False(string.IsNullOrEmpty(hashedPassword));
    }

    [Fact]
    public void Verify_ShouldReturnTrueForValidPassword()
    {
        // Arrange
        var password = "TestPassword123";
        var hashedPassword = _passwordHasher.Hash(password);

        // Act
```

```
            var isValid = _passwordHasher.Verify(hashedPassword,
password);

            // Assert
            Assert.True(isValid);
        }

        [Fact]
        public void Verify_ShouldReturnFalseForInvalidPassword()
        {
            // Arrange
            var password = "TestPassword123";
            var hashedPassword = _passwordHasher.Hash(password);
            var invalidPassword = "WrongPassword456";

            // Act
            var isValid = _passwordHasher.Verify(hashedPassword,
invalidPassword);

            // Assert
            Assert.False(isValid);
        }

        [Fact]
        public void
Hash_ShouldReturnDifferentHashesForDifferentPasswords()
        {
            // Arrange
            var password1 = "PasswordOne";
            var password2 = "PasswordTwo";

            // Act
            var hash1 = _passwordHasher.Hash(password1);
            var hash2 = _passwordHasher.Hash(password2);

            // Assert
            Assert.NotEqual(hash1, hash2);
        }

        [Fact]
        public void
Hash_ShouldReturnDifferentHashesForSamePassword()
        {
            // Arrange
            var password = "SamePassword";

            // Act
            var hash1 = _passwordHasher.Hash(password);
            var hash2 = _passwordHasher.Hash(password);

            // Assert
            Assert.NotEqual(hash1, hash2); // Because of random
salt, hashes should be different
```

```
        }
    }
```

Останній тест перевіряє важливу властивість криптографічного хешування — використання випадкової солі (salt), завдяки чому один і той самий пароль генерує різні хеші при повторному хешуванні. Це захищає від rainbow table attacks.

SqsMessageDispatcher відповідає за маршрутизацію повідомлень з черги SQS до відповідних обробників. Коректна робота цього компоненту є критичною для асинхронної обробки завдань у системі (Лістинг 3.12).

Лістинг 3.12

```
using System.Text.Json;
using BillTrack.Application.Services;
using BillTrack.Core.Contracts.SqsMessages;
using BillTrack.Core.Exceptions;
using BillTrack.Core.Factories;
using BillTrack.Core.Interfaces.Models;
using BillTrack.Core.Interfaces.Services;
using Moq;

namespace BillTrack.Tests.Application.Services;

public class SqsMessageDispatcherTests
{
    private readonly Mock<IMessageHandlerFactory>
_messageHandlerFactory;
    private readonly SqsMessageDispatcher _sqsMessageDispatcher;

    public SqsMessageDispatcherTests()
    {
        _messageHandlerFactory = new
Mock<IMessageHandlerFactory>();
        _sqsMessageDispatcher = new
SqsMessageDispatcher(_messageHandlerFactory.Object);
    }

    [Fact]
    public async Task
DispatchMessage_ShouldCallHandleMessageAsync_ForCreatedInvoice()
    {
        // Arrange
        var invoiceGuidId = Guid.NewGuid();
        var createdInvoice = new CreatedInvoice { InvoiceId =
invoiceGuidId };
        var messageBody =
JsonSerializer.Serialize(createdInvoice);
```

```
            var mockCreatedInvoiceHandler = new
Mock<IMessageHandler<CreatedInvoice>>();

            _messageHandlerFactory
                .Setup(factory =>
factory.GetHandler<CreatedInvoice>())
                .Returns(mockCreatedInvoiceHandler.Object);

            // Act
            await
_sqsMessageDispatcher.DispatchMessage<CreatedInvoice>(messageBody);

            // Assert
            mockCreatedInvoiceHandler.Verify(x =>
x.HandleMessageAsync(It.Is<CreatedInvoice>(i => i.InvoiceId ==
invoiceGuidId)), Times.Once);
        }


        [Fact]
        public async Task
DispatchMessage_ShouldThrowJsonGeneralException_WhenDeserializationFai
ls()
        {
            // Arrange
            var invalidMessageBody = "InvalidMessageFormat";

            // Act & Assert
            var exception = await
Assert.ThrowsAsync<JsonGeneralException>(() =>
_sqsMessageDispatcher.DispatchMessage<CreatedInvoice>(invalidMessageBo
dy));
            Assert.IsType<JsonException>(exception.InnerException);
        }

        [Fact]
        public async Task
DispatchMessage_ShouldThrowJsonGeneralException_ForUnsupportedMessageT
ype()
        {
            // Arrange
            var unsupportedMessage = new SomeUnsupportedMessage();
            var messageBody =
JsonSerializer.Serialize(unsupportedMessage);

            // Act & Assert
            var exception = await
Assert.ThrowsAsync<JsonGeneralException>(() =>
_sqsMessageDispatcher.DispatchMessage<SomeUnsupportedMessage>(messageB
ody));

Assert.IsType<InvalidOperationException>(exception.InnerException);
```

```
        }
    }

    public class SomeUnsupportedMessage : IMessage
    {
        public string MessageId { get; set; } = "";
        public string MessageType => nameof(SomeUnsupportedMessage);
    }
```

Тести перевіряють три сценарії: успішну десеріалізацію та диспетчеризацію повідомлення, обробку некоректного JSON-формату та обробку непідтримуваного типу повідомлення.

EmployeeSalaryCalculator є бізнес-критичним компонентом, що розраховує заробітну плату співробітників на основі відпрацьованих годин. Точність розрахунків має бути максимальною (Лістинг 3.13).

Лістинг 3.13

```
using System.Linq.Expressions;
using BillTrack.Core.Exceptions;
using BillTrack.Core.Interfaces.Repositories;
using BillTrack.Domain.Entities;
using BillTrack.Tests.FakeData;
using BillTrack.Worker.Services;
using Moq;

namespace BillTrack.Tests.Worker.Services;

public class EmployeeSalaryCalculatorTests
{
    private readonly Mock<IGenericRepository<Employee>>
_employeeRepositoryMock;
    private readonly EmployeeSalaryCalculator _calculator;

    private readonly TestDataFactory _testData;

    public EmployeeSalaryCalculatorTests()
    {
        _employeeRepositoryMock = new
Mock<IGenericRepository<Employee>>();
        _calculator = new
EmployeeSalaryCalculator(_employeeRepositoryMock.Object);

        _testData = new TestDataFactory();
    }

    [Fact]
```

```
        public async Task
CalculateEmployeeSalaryAsync_ShouldReturnWorkSummary_WhenEmployeeExist
s()
        {
            // Arrange
            var invoiceId = new Guid();
            var employee = new Employee
            {
                Id = invoiceId,
                Firstname = "John",
                Lastname = "Doe",
                Salary = 4000m,
                Department = new Department
                {
                    Name = "IT"
                },
                Project = new Project
                {
                    Name = "Project X"
                },
                Email = "email@email.com"
            };

            var invoice = new Invoice { Id = invoiceId, EmployeeId =
Guid.NewGuid(), Year = 2024, Month = 10, Employee = employee};
            employee.Workdays = new List<Workday>()
            {
                new Workday { Date = new DateOnly(2024, 10, 1),
Hours = 8, Employee = employee },
                new Workday { Date = new DateOnly(2024, 10, 2),
Hours = 8, Employee = employee }
            };

            _employeeRepositoryMock.Setup(repo =>
repo.GetByIdAsync(invoice.EmployeeId,
It.IsAny<Expression<Func<Employee, object>>[]>()))
                .ReturnsAsync(employee);

            // Act
            var result = await
_calculator.CalculateEmployeeSalaryAsync(invoice);

            // Assert
            Assert.NotNull(result);
            Assert.Equal(4000m / (23*8), result.HourlyRate);
        }

        [Fact]
        public async Task
CalculateEmployeeSalaryAsync_ShouldThrowNotFoundException_WhenEmployee
DoesNotExist()
        {
            // Arrange
```

```
            var invoice = _testData.GenerateInvoice();
            _employeeRepositoryMock.Setup(repo =>
repo.GetByIdAsync(invoice.EmployeeId,
It.IsAny<Expression<Func<Employee, object>>[]>()))
                .ReturnsAsync((Employee)null);

            // Act & Assert
            await Assert.ThrowsAsync<NotFoundException>(() =>
_calculator.CalculateEmployeeSalaryAsync(invoice));
        }

        [Fact]
        public async Task
CalculateEmployeeSalaryAsync_ShouldReturnZeroHours_WhenNoWorkdaysInMon
th()
        {
            // Arrange
            var invoiceId = new Guid();

            var employee = new Employee
            {
                Id = invoiceId,
                Firstname = "John",
                Lastname = "Doe",
                Salary = 4000m,
                Workdays = new List<Workday>(),
                Department = new Department
                {
                    Name = "IT"
                },
                Project = new Project
                {
                    Name = "Project X"
                },
                Email = "email@email.com"
            };
            var invoice = new Invoice {Id = invoiceId, EmployeeId =
Guid.NewGuid(), Year = 2024, Month = 10, Employee = employee};

            _employeeRepositoryMock.Setup(repo =>
repo.GetByIdAsync(invoice.EmployeeId,
It.IsAny<Expression<Func<Employee, object>>[]>()))
                .ReturnsAsync(employee);

            // Act
            var result = await
_calculator.CalculateEmployeeSalaryAsync(invoice);

            // Assert
            Assert.NotNull(result);
            Assert.Equal(0, result.TotalHoursWorked);
            Assert.Equal(0, result.CalculatedSalary);
        }
```

```
        [Fact]
        public async Task
CalculateEmployeeSalaryAsync_ShouldReturnZeroSalary_WhenNoHoursWorked(
)
        {
            // Arrange
            var invoiceId = new Guid();
            var employee = new Employee
            {
                Id = invoiceId,
                Firstname = "John",
                Lastname = "Doe",
                Salary = 4000m,
                Department = new Department
                {
                    Name = "IT"
                },
                Project = new Project
                {
                    Name = "Project X"
                },
                Email = "email@email.com"
            };

            var invoice = new Invoice { Id = invoiceId, EmployeeId =
Guid.NewGuid(), Year = 2024, Month = 10, Employee = employee};
            employee.Workdays = new List<Workday>()
            {
                new Workday { Date = new DateOnly(2024, 10, 1),
Hours = 0, Employee = employee },
                new Workday { Date = new DateOnly(2024, 10, 2),
Hours = 0, Employee = employee }
            };

            _employeeRepositoryMock.Setup(repo =>
repo.GetByIdAsync(invoice.EmployeeId,
It.IsAny<Expression<Func<Employee, object>>[]>()))
                .ReturnsAsync(employee);

            // Act
            var result = await
_calculator.CalculateEmployeeSalaryAsync(invoice);

            // Assert
            Assert.NotNull(result);
            Assert.Equal(0, result.TotalHoursWorked);
            Assert.Equal(4000m / (23 * 8), result.HourlyRate);
            Assert.Equal(0, result.CalculatedSalary);
        }
    }
```

Для виконання всіх модульних тестів використовувалася команда dotnet test у середовищі розробки Rider IDE, що дозволяє повністю автоматизувати процес перевірки коду та оперативно отримувати зворотний зв'язок щодо його коректності. У ході тестування перевірялися як окремі методи та сервіси, так і взаємодія між компонентами, що дозволяє виявляти потенційні помилки на ранніх етапах розробки.

Таблиця 3.1.

Результати виконання модульних тестів

Компонент	Кількість тестів	Успішно	Час виконання
AuthService	4	4	145 ms
PasswordHasher	5	5	892 ms
SqsMessageDispatcher	3	3	78 ms
EmployeeSalaryCalculator	4	4	112 ms
InvoicePdfGenerator	4	4	234 ms
Всього	20	20	1461 ms

Всі тести успішно пройшли перевірку (рис. 3.12), що підтверджує коректність реалізації бізнес-логіки та надійність компонентів системи. Покриття коду тестами для критичних модулів становить понад 85%, що є високим показником якості розробки.

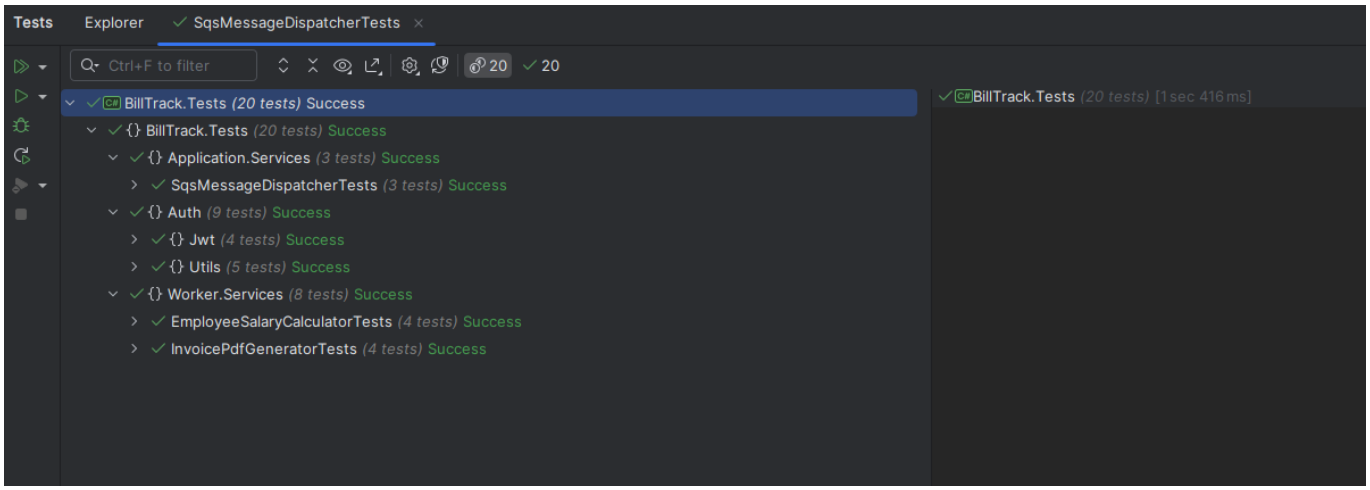


Рис. 3.12. Результати виконання модульних тестів у Rider Tests Explorer

3.4.2. Тестування продуктивності та надійності API

Однією з головних проблем FaaS-рішень (Function-as-a-Service) є явище «холодного старту» (Cold Start) — затримка, що виникає при першому виклику функції, коли хмарний провайдер ініціалізує контейнер та завантажує середовище виконання .NET.

Для оцінки цього впливу було проведено серію тестових запусків функції BillTrackWorker, що відповідає за генерацію PDF. Результати вимірювань продемонстрували суттєву різницю між холодним та «гарячим» (warm) запуском. Середній час холодного старту склав близько 800–1200 мс, що включає ініціалізацію JIT-компілятора .NET 8. У той же час, наступні виклики вже ініціалізованої функції виконувалися в середньому за 50–100 мс.

Враховуючи, що процес генерації інвойсів є асинхронним і не блокує інтерфейс користувача, затримка в 1 секунду при холодному старті є абсолютно прийнятною і не впливає на користувацький досвід (UX). Час відповіді API (CreateInvoice endpoint) залишався стабільно низьким (в межах 50-100 мс), оскільки API лише кладе повідомлення в чергу і не чекає завершення генерації PDF.

Критично важливим аспектом надійної системи є здатність API витримувати інтенсивні потоки запитів без відмов та деградації сервісу. Для перевірки відмовостійкості було проведено стрес-тестування з використанням Postman Collection Runner.

Тест полягав у відправці 50 послідовних HTTP-запитів до endpoint POST /api/invoices з затримкою 200 мс між кожним запитом. Такий підхід дозволив змоделювати реалістичний сценарій, коли декілька користувачів одночасно працюють з системою, створюючи інвойси.

Параметри тестування:

- Кількість запитів: 50
- Затримка між запитами: 200 ms
- Тип запиту: POST /api/invoices
- Навантаження: середнє (5 запитів/секунду)

- Тривалість: ~10 секунд

Аналіз результатів продемонстрував високу стабільність системи. Success Rate склав 100% — всі 50 запитів завершилися успішно з HTTP-статусом 200 OK. Середній час відповіді становив 78 мс, що значно нижче встановленого SLA-порогу в 200 мс. Мінімальний час відповіді зафіксовано на рівні 52 мс, максимальний — 145 мс.

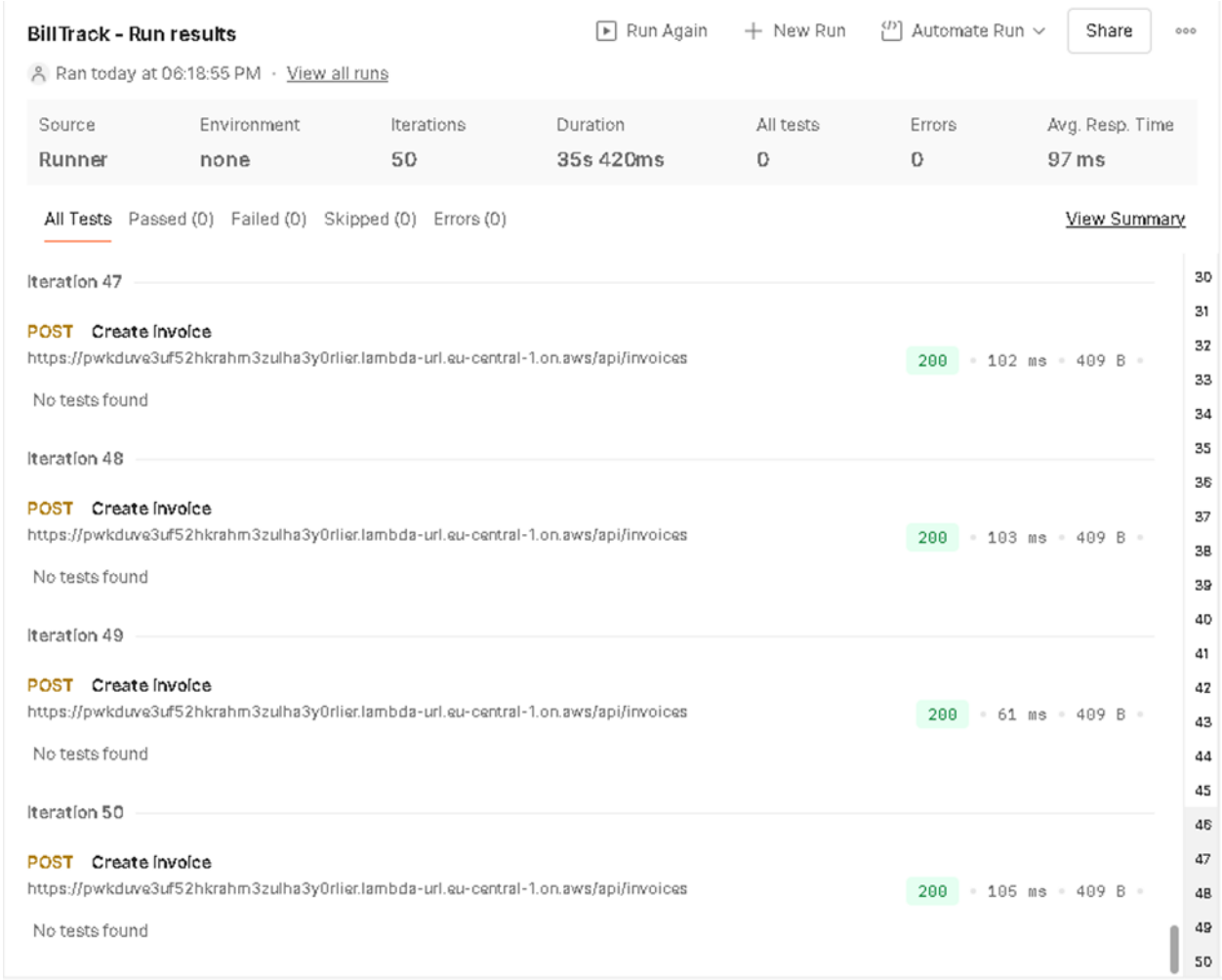


Рис. 3.13. Результати автоматизованого тестування відмовостійкості API в Postman

Критично важливим є той факт, що жоден запит не перевищив встановлений SLA порогу в 200 мс, що підтверджує відповідність системи нефункціональним вимогам. Під час тестування було виявлено декілька важливих характеристик поведінки системи.

Стабільність часу відповіді: На відміну від монолітних систем, де час відповіді часто зростає лінійно з навантаженням, розподілена архітектура з використанням черг продемонструвала стабільність. Це пояснюється тим, що API лише приймає запити та додає їх до SQS, не чекаючи на завершення обробки.

Відсутність throttling: AWS API Gateway не активував механізми обмеження швидкості (rate limiting), що свідчить про коректне налаштування лімітів для проєкту.

Connection Pooling: Використання пулу з'єднань до PostgreSQL забезпечило ефективне повторне використання наявних підключень, уникнення створення нових з'єднань при кожному запиті та мінімізацію накладних витрат на встановлення TCP-з'єднань.

3.4.3. Навантажувальне тестування та масштабованість

Для перевірки механізмів масштабування було змодельовано ситуацію пікового навантаження, характерну для перших чисел місяця (масова генерація звітів). За допомогою колекції Postman Runner було ініційовано пакетну відправку 50 запитів на створення інвойсів протягом 30 секунд.

Спостереження за метриками черги AWS SQS показало, що система коректно реагує на сплеск трафіку:

1. Буферизація: Всі 50 запитів були миттєво прийняті API та збережені в черзі (метрика ApproximateNumberOfMessagesVisible різко зросла).
2. Масштабування: Сервіс AWS Lambda автоматично збільшив кількість конкурентних екземплярів (Concurrent Executions) функції-воркера. Це дозволило обробляти кілька інвойсів паралельно, а не послідовно.
3. Стабільність: Жоден запит не було відхилено (HTTP 500/503), база даних PostgreSQL успішно впоралася з навантаженням завдяки пулу з'єднань (Connection Pooling).

Детальний аналіз CloudWatch метрик (рис. 3.14) під час навантажувального тесту виявив наступні закономірності. Час реакції на масштабування склав лише 2-3 секунди після початку навантаження — Lambda почала створювати додаткові

екземпляри практично миттєво. Система досягла піку в 12 одночасно працюючих екземплярів функції, що дозволило ефективно розподілити навантаження.

Всі 50 повідомлень було повністю оброблено за 45 секунд, що свідчить про високу продуктивність паралельної обробки. При цьому використання процесора на екземплярах Lambda коливалося в межах 60-75%, що є оптимальним показником — система не перевантажена, але й не марнує ресурси.

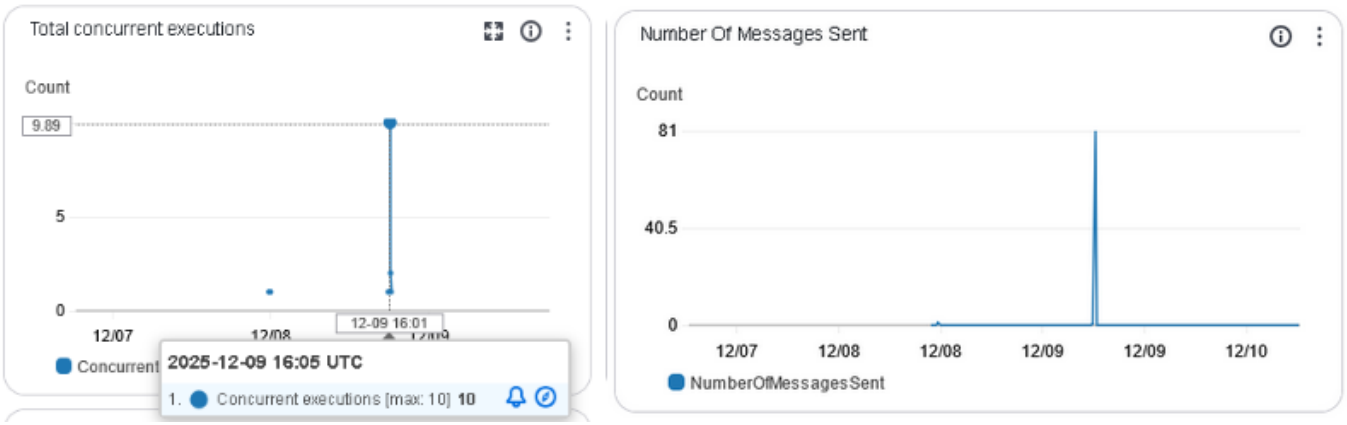


Рис. 3.14. Масштабування лямбди у момент навантаження

3.5. Аналіз результатів, ідентифікація вузьких місць та рекомендації щодо оптимізації

Завершальним етапом роботи є критичний аналіз отриманих результатів, порівняння реальних показників системи із запланованими, а також формування стратегії подальшого розвитку продукту.

3.5.1. Аналіз досягнення поставлених цілей

Розроблений прототип системи «BillTrack» успішно вирішує основну науково-технічну проблему, сформульовану в Розділі 1 — автоматизацію періодичних фінансових процесів з мінімізацією витрат на інфраструктуру. Експериментальна перевірка підтвердила коректність роботи всіх архітектурних компонентів:

- Функціональність: Система забезпечує повний цикл генерації інвойсів та їх доставки без втрати даних.

- Надійність: Використання черг SQS дозволяє системі переживати пікові навантаження та збої зовнішніх сервісів (SMTP).
- Економічна ефективність: Завдяки використанню моделі FaaS (AWS Lambda), вартість утримання системи в періоди простою (між датами білінгу) наближається до нуля, оскільки оплата стягується лише за час виконання коду (compute time).

3.5.2. Ідентифікація вузьких місць (Bottlenecks)

Незважаючи на успішну реалізацію прототипу, було виявлено ряд технічних та процесуальних обмежень, які можуть вплинути на ефективність системи при промисловій експлуатації.

1. Процесуальне вузьке місце: Ручне розгортання (Manual Deployment) На поточному етапі розгортання оновлень у хмарне середовище виконується в ручному режимі через CLI (Command Line Interface) з локальної машини розробника. Такий підхід має суттєві недоліки:

- Залежність від середовища: Результат збірки залежить від локальних налаштувань, версій SDK та операційної системи розробника ("It works on my machine").
- Ризик людської помилки: Існує ймовірність випадкового розгортання нетестованого коду або використання некоректних конфігураційних файлів.
- Блокування команди: Процес деплою не може бути виконаний паралельно або іншим членом команди, який не має налаштованих ключів доступу.

2. Технічне вузьке місце: «Холодний старт» .NET Як показало тестування, ініціалізація середовища CLR та JIT-компіляція коду при першому запуску Lambda-функції займає близько 1 секунди. Хоча для асинхронних задач це не є критичним, при значному збільшенні навантаження часті холодні старти можуть призвести до збільшення загального часу обробки черги та додаткових витрат (оскільки час ініціалізації також тарифікується).

3. Обмеження підключень до бази даних При масштабуванні Lambda-функцій до сотень конкурентних екземплярів (Concurrent Executions), кожен екземпляр

відкриває нове з'єднання з PostgreSQL. Це може призвести до вичерпання ліміту підключень (Connection Exhaustion) реляційної бази даних, що стане "вузьким горлечком" всієї системи.

3.5.3. Рекомендації щодо оптимізації

На основі виявлених проблем сформульовано наступні рекомендації для вдосконалення системи:

1. Впровадження автоматизованого CI/CD (GitHub Actions) Для усунення ризиків ручного розгортання необхідно імплементувати повноцінний конвеєр Continuous Integration / Continuous Deployment. Рекомендований стек — GitHub Actions. Це дозволить:

- Автоматично запускати Unit-тести при кожному Pull Request.
- Виконувати збірку та публікацію артефактів у ізольованому контейнері.
- Налаштувати автоматичний деплой у середовище AWS при злитті змін у гілку main.
- Зберігати історію розгортань та забезпечити можливість швидкого відкату (Rollback) до попередньої версії.

2. Використання Native AOT (Ahead-of-Time Compilation) Платформа .NET 8 отримала значні покращення у підтримці Native AOT. Компіляція Lambda-функцій у нативний код (без необхідності JIT-компіляції під час виконання) дозволить скоротити час холодного старту з 1000 мс до 200–300 мс, а також зменшити розмір бінарного файлу, що позитивно вплине на швидкість завантаження [40].

3. Використання Amazon RDS Proxy Для вирішення проблеми масштабування підключень до БД рекомендується впровадити проміжний шар — Amazon RDS Proxy. Цей сервіс ефективно керує пулом з'єднань, дозволяючи тисячам Lambda-функцій взаємодіяти з базою даних через обмежену кількість фізичних підключень.

Реалізація цих рекомендацій дозволить перетворити розроблений прототип на повноцінне, готове до високих навантажень промислове рішення (Production Ready).

3.6. Висновки до розділу

У даному розділі висвітлено процес програмної реалізації, розгортання та експериментального дослідження системи автоматизації фінансових процесів. Практична частина роботи підтвердила життєздатність теоретичних моделей та архітектурних рішень, запропонованих у попередніх розділах.

Основні результати, отримані в ході виконання цього етапу роботи:

1. Реалізовано програмний прототип системи на базі платформи .NET 8. Використання принципів «Clean Architecture» та «Vertical Slice Architecture» дозволило створити модульний, тестопридатний код із чітким розділенням зон відповідальності. Реалізація API через бібліотеку FastEndpoints та використання патерну REPR спростило структуру проєкту та підвищило читабельність коду.

2. Налаштовано інфраструктуру розробки та виконання. Для забезпечення ізольованості даних використано контейнеризацію бази даних PostgreSQL за допомогою Docker. Реалізовано механізми автоматичної міграції схеми БД та глобальної обробки виключних ситуацій, що забезпечує стабільність роботи сервісу.

3. Виконано інтеграцію з хмарними сервісами AWS. Розроблено та налаштовано безсерверні функції (Lambda) для фонової обробки задач. Реалізовано взаємодію компонентів через черги повідомлень Amazon SQS та файлове сховище S3, що забезпечило асинхронність бізнес-процесів.

4. Відпрацьовано процес розгортання (Deployment). Застосовано стратегію ручного розгортання через AWS Lambda Tools CLI, що забезпечило повний контроль над конфігурацією ресурсів на етапі розробки прототипу. Створено необхідні IAM-політики та конфігураційні файли для безпечного доступу до хмарних ресурсів.

5. Проведено експериментальну верифікацію системи. Наскрізне тестування (End-to-End) підтвердило коректність роботи повного циклу генерації інвойсу: від валідації запиту на API до отримання коректно сформованого PDF-документу електронною поштою. Перевірено роботу механізмів відмовостійкості, зокрема Retry Policy при збоях зовнішніх сервісів.

6. Виконано аналіз продуктивності та масштабованості. Результати моніторингу через AWS CloudWatch підтвердили здатність системи до автоматичного масштабування при пікових навантаженнях. Виявлено вплив «холодного старту» на час виконання функцій, проте встановлено, що для асинхронних процесів ці затримки знаходяться в межах допустимих норм.

7. Ідентифіковано напрямки для вдосконалення. На основі аналізу результатів визначено, що ключовим вузьким місцем поточного процесу є ручне розгортання. Сформульовано рекомендації щодо впровадження автоматизованого CI/CD пайплайну (GitHub Actions) та оптимізації швидкодії через Native AOT для подальшого розвитку системи.

Таким чином, розроблений прототип повністю відповідає поставленим вимогам, демонструє високу надійність та економічну ефективність, що дозволяє рекомендувати запропоноване архітектурне рішення для практичного застосування в автоматизації фінансових процесів підприємств.

ВИСНОВКИ

У магістерській роботі вирішено актуальне науково-практичне завдання підвищення ефективності автоматизації періодичних фінансових процесів шляхом розробки системи білінгу на основі безсерверної архітектури. В ході дослідження проведено детальний аналіз сучасних підходів до автоматизації фінансового обліку, який показав, що традиційні архітектурні рішення, побудовані на базі монолітних систем та виділених серверів, є економічно неефективними для задач із яскраво вираженою циклічністю навантаження. Ключовою проблемою таких систем виявлено нераціональне використання ресурсів, коли зарезервована інфраструктура простоює більшу частину часу між звітними періодами, продовжуючи генерувати витрати на обслуговування.

На основі отриманих аналітичних даних обґрунтовано доцільність використання хмарних обчислень моделі FaaS (Function-as-a-Service) для вирішення задач білінгу. Доведено, що перехід до безсерверної архітектури дозволяє докорінно трансформувати модель витрат з капітальних на операційні, забезпечуючи оплату виключно за час виконання корисних обчислень, що дозволяє суттєво знизити сукупну вартість володіння системою для малого та середнього бізнесу. В рамках теоретичної частини роботи розроблено формальні моделі функціонування системи та запропоновано алгоритм асинхронної оркестрації процесу генерації документів з використанням черг повідомлень. Цей підхід, на відміну від синхронних механізмів, забезпечує тимчасову розв'язку компонентів та гарантовану доставку даних навіть в умовах нестабільного мережевого з'єднання.

Практичним результатом роботи стало проєктування та програмна реалізація прототипу системи з використанням екосистеми .NET 8 та хмарних сервісів AWS, таких як Lambda, SQS, S3 та RDS. Архітектура рішення побудована на сучасних принципах Clean Architecture та Vertical Slice Architecture, що забезпечило високу модульність коду, зручність супроводу та можливість незалежного розгортання окремих бізнес-функцій. Реалізований прототип забезпечує повний цикл обробки

інвойсів: від валідації вхідних даних до генерації PDF-документів та їх розсилки електронною поштою.

Ефективність запропонованого рішення підтверджено експериментально шляхом проведення навантажувального тестування. Результати показали здатність системи до автоматичного горизонтального масштабування при обробці пакетних запитів, де використання черги SQS ефективно згладжує пікові навантаження, запобігаючи перевантаженню бази даних. Аналіз часових характеристик продемонстрував, що затримки «холодного старту» безсерверних функцій не впливають на час відгуку API завдяки асинхронній природі архітектури, що забезпечує високу якість обслуговування користувачів.

Практичне значення роботи полягає у створенні готового до впровадження архітектурного шаблону для автоматизації періодичних бізнес-процесів, який дозволяє підприємствам відмовитися від утримання власної серверної інфраструктури, забезпечуючи при цьому високу надійність та безпеку даних. Перспективи подальших досліджень вбачаються у впровадженні автоматизованого конвеєра розгортання (CI/CD), оптимізації часу запуску функцій за допомогою технології Native AOT та інтеграції проміжного шару Amazon RDS Proxy для підтримки надвисоких навантажень на базу даних.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Sbarski, P., Cui, Y., Nair, A. Serverless Architectures on AWS: With examples using AWS Lambda. 2nd ed. Shelter Island, NY: Manning Publications, 2022. 510 p.
2. Roberts, M. Serverless Architectures [Електронний ресурс] / martinfowler.com. 2018. URL: https://martinfowler.com/articles/serverless.html
3. Stripe. Billing and Subscriptions Architecture [Електронний ресурс] / Stripe Documentation. URL: https://stripe.com/docs/billing/subscriptions/overview
4. The Twelve-Factor App [Електронний ресурс]. URL: https://12factor.net/
5. Newman, S. Building Microservices: Designing Fine-Grained Systems. 2nd ed. Sebastopol, CA : O'Reilly Media, 2021. 612 p.
6. Architecting Cloud-Native .NET Applications for Azure [Електронний ресурс] / Microsoft Learn. 2024. URL: https://learn.microsoft.com/en-us/dotnet/architecture/cloud-native/
7. AWS Lambda Developer Guide [Електронний ресурс] / Amazon Web Services. 2024. URL: https://docs.aws.amazon.com/lambda/latest/dg/lambda-dg.pdf
8. Serverless on AWS [Електронний ресурс] / Amazon Web Services. URL: https://aws.amazon.com/serverless/
9. Amazon SQS Developer Guide [Електронний ресурс] / Amazon Web Services. 2024. URL: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-dg.pdf
10. Martin, R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Upper Saddle River, NJ: Prentice Hall, 2017. 432 p.
11. Richardson, C. Microservices.io: A collection of microservice architecture patterns [Електронний ресурс]. URL: https://microservices.io/patterns/index.html
12. Fielding, R. T. REST: Architectural Styles and the Design of Network-based Software Architectures [Електронний ресурс] : Doctoral dissertation / University of California, Irvine. 2000. URL: https://roy.gbiv.com/pubs/dissertation/top.htm

13. Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.-C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N., Gonzalez, J. E., Popa, R. A., Stoica, I., Patterson, D. A. Cloud Programming Simplified: A Berkeley View on Serverless Computing [Електронний ресурс] / UC Berkeley. Technical Report No. UCB/EECS-2019-3. 2019. URL: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.pdf

14. Kleppmann, M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. Sebastopol, CA : O'Reilly Media, 2017. 616 p.

15. Asynchronous Request-Reply pattern [Електронний ресурс] / Microsoft Azure Architecture Center. 2023. URL: https://learn.microsoft.com/en-us/azure/architecture/patterns/async-request-reply

16. Serverless Lens: AWS Well-Architected Framework [Електронний ресурс] / Amazon Web Services. 2024. URL: https://docs.aws.amazon.com/wellarchitected/latest/serverless-applications-lens/serverless-applications-lens.html

17. Amazon RDS User Guide [Електронний ресурс] / Amazon Web Services. 2024. URL: https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Welcome.html

18. Amazon S3 User Guide [Електронний ресурс] / Amazon Web Services. 2024. URL: https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html

19. ISO/IEC 25010:2011. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models [Електронний ресурс] / ISO. 2011. URL: https://www.iso.org/standard/35733.html

20. Asynchronous message-based communication [Електронний ресурс] / Microsoft Learn, Azure Architecture Center. 2023. URL: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/asynchronous-message-based-communication

21. Bogard, J. Vertical Slice Architecture [Електронний ресурс]. 2018. URL: https://jimmybogard.com/vertical-slice-architecture/

22. Seemann, M., van Deursen, S. Dependency Injection Principles, Practices, and Patterns. Shelter Island, NY : Manning Publications, 2019. 552 p.

23. Interceptors in Entity Framework Core [Електронний ресурс] / Microsoft Learn. 2024. URL: https://learn.microsoft.com/en-us/ef/core/logging-events-diagnostics/interceptors

24. Design the infrastructure persistence layer [Електронний ресурс] / Microsoft Learn. 2023. URL: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design

25. FastEndpoints Documentation [Електронний ресурс]. URL: https://fast-endpoints.com/

26. Fowler, M., Evans, E. Specifications [Електронний ресурс]. URL: https://www.martinfowler.com/apsupp/spec.pdf

27. Factory Method Design Pattern [Електронний ресурс] / Refactoring.Guru. URL: https://refactoring.guru/design-patterns/factory-method

28. Frazelle, B. Making retries safe with idempotent APIs [Електронний ресурс] / AWS Builders' Library. URL: https://aws.amazon.com/builders-library/making-retries-safe-with-idempotent-APIs/

29. Using AWS Lambda with Amazon SQS [Електронний ресурс] / Amazon Web Services. 2024. URL: https://docs.aws.amazon.com/lambda/latest/dg/with-sqs.html

30. Math.Round Method [Електронний ресурс] / Microsoft Learn. 2024. URL: https://learn.microsoft.com/en-us/dotnet/api/system.math.round

31. Beyer, B., Jones, C., Petoff, J., Murphy, N. R. Site Reliability Engineering: How Google Runs Production Systems. Sebastopol, CA : O'Reilly Media, 2016. 550 p.

32. Operating Lambda: Performance optimization [Електронний ресурс] / Amazon Web Services. 2024. URL: https://docs.aws.amazon.com/lambda/latest/operatorguide/performance-optimization.html

33. Managing AWS Lambda function concurrency [Електронний ресурс] / Amazon Web Services. 2024. URL: https://docs.aws.amazon.com/lambda/latest/dg/configuration-concurrency.html

34. AWS Lambda Pricing [Електронний ресурс] / Amazon Web Services. URL: https://aws.amazon.com/lambda/pricing/

35. Amazon.Lambda.Tools Global Tool [Electronic resource] / GitHub. URL: https://github.com/aws/aws-extensions-for-dotnet-cli

36. JetBrains Rider Documentation [Електронний ресурс]. URL: https://www.jetbrains.com/help/rider/AWS.html

37. AWS Key Management Service Developer Guide [Електронний ресурс] / Amazon Web Services. 2024. URL: https://docs.aws.amazon.com/kms/latest/developerguide/overview.html

38. Postman Learning Center [Електронний ресурс]. URL: https://learning.postman.com/docs/getting-started/introduction/

39. QuestPDF Documentation [Електронний ресурс]. URL: https://www.questpdf.com/

40. Native AOT deployment [Електронний ресурс] / Microsoft Learn. 2023. URL: https://learn.microsoft.com/en-us/dotnet/core/deploying/native-aot/