МІНІСТЕРСТВО ОСІТИ І НАУКИ УКРАЇНИ

Харківський національний університет радіоелектроніки

Факультет _____Комп'ютерних наук_____
(повна назва)

Кафедра _____ Програмної Інженерії_____
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти _____другий (магістерський)_____

Дослідження архітектурних підходів до забезпечення приватності конфіденційних даних у хмарному середовищі
(тема)

Виконав:

Студент 2 курсу, групи _____ІПЗм-22-1____

_____Трибух А.О_____
(прізвище, ініціали)

Спеціальність 121 – Інженерія програмного__
_____забезпечення_____
(код і повна назва спеціальності)

Тип програми ____освітньо-наукова_____
(повна назва освітньої програми)

Керівник __доцент кафедри ПІ Каук В.І__
(посада, прізвище, ініціали)

Допускається до захисту
Зав. кафедри _____ З.В.Дудар
підпис (прізвище, ініціали)

2024 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
Кафедра програмної інженерії
Рівень вищої освіти другий (магістерський)
Спеціальність 121 – Інженерія програмного забезпечення
Тип програми: освітньо-наукова програма
Освітня програма Інженерія програмного забезпечення
(шифр і назва)

ЗАТВЕРДЖУЮ:

Зав. кафедри_____

«_____»_____20 ___ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Трибуху Арсенію Олексійовичу

1. Тема роботи: Дослідження архітектурних підходів до забезпечення приватності конфіденційних даних у хмарному середовищ

затверджена наказом по університету від "29" березня 2024 р. № 250Ст

2. Термін подання студентом роботи до екзаменаційної комісії «_21_» _06_2024 р.

3. Вихідні дані до роботи: дослідити вплив різних типів архітектур програмних систем на безпеку даних при розгортанні в хмарному середовищі Для дослідження використовувати два різних хмарних провайдери. Створити прототип бекенду програмної системи з використанням платформи .NET, мови програмування C#, фреймворку ASP.NETCore. Використати як реляційну так і нереляційну бази даних. Провести порівняння продуктивності та безпеки системи. Зробити висновки з результатів.

4. Перелік питань, що потрібно опрацювати в роботі: аналіз предметної галузі, підготовка до експерименту, підготовка до експерименту, створення прототипу програмної системи, проведення експерименту, висновки, перелік посилань, додатки.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання	Примітка
1	Аналіз предметної галузі та постановка задачі	10.04.2024	виконано
2	Створення прототипів програмної системи з використанням різних архітектур	20.05.2024	виконано
3	Проведення дослідження	03.06.2024	виконано
4	Аналіз результатів дослідження та формування висновків	05.06.2024	виконано
5	Підготовка пояснювальної записки	10.06.2024	виконано
6	Перевірка пояснювальної записки керівником, підготовка роботи для проходження перевірки на антиплагіат	12.06.2024	виконано
7	Оцінка роботи рецензентом, отримання відзиву від керівника атестаційної роботи, попередній захист роботи та проходження нормо контролю	15.06.2024	виконано
8	Здача роботи у електронний архів, допуск роботи до захисту завідувачем кафедри	18.06.2024	виконано
9	Захист кваліфікаційної роботи	21.06.2024	виконано

Дата видачі завдання «29 » березня 2024 р.

Студент _____ Трибух А.О

Керівник роботи _____ доц. Каук В.І

підпис (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Робота містить: 93 с., 15 рис., 4 табл., 28 джер.

БЕЗПЕКА ДАНИХ, БАГАТОФАКТОРНА АУТЕНТИФІКАЦІЯ, МОДЕЛЬ ZERO TRUST, MICROSOFT AZURE, ПРОГРАМНА АРХІТЕКТУРА, ТЕСТУВАННЯ НА ПРОНИКНЕННЯ, ХМАРНЕ СЕРЕДОВИЩЕ, ШИФРУВАННЯ, .NET, ASP NET CORE, AWS, DOCKER.

Об'єктом дослідження є архітектурні підходи для забезпечення безпеки даних у програмних системах в хмарному середовищі.

Метою роботи є підвищення рівня безпеки даних у хмарних додатках шляхом впровадження різних архітектурних підходів, включаючи модель Zero Trust.

Методи розробки базуються на таких технологіях, як ASP.NET Core, .NET, Docker, AWS, Microsoft Azure.

В результаті роботи було досліджено різні архітектурні підходи до забезпечення безпеки даних, проведено порівняльний аналіз їхньої ефективності та впливу на продуктивність системи. Було розроблено прототип програмної системи для роботи з даними пацієнтів, що включає багаторівневий захист даних, багатоетапну аутентифікацію та постійний моніторинг активності. Проведено тестування системи на проникнення та інші безпекові тести для оцінки рівня захисту, забезпеченого різними архітектурними підходами. На основі отриманих результатів було надано рекомендації щодо підвищення безпеки даних у хмарних додатках.

ASP NET CORE, AWS, CLOUD ENVIRONMENT, DATA SECURITY, DOCKER, ENCRYPTION, MICROSOFT AZURE, MULTI-FACTOR AUTHENTICATION, .NET, PENETRATION TESTING, SOFTWARE ARCHITECTURE, ZERO TRUST MODEL.

The object of the study is architectural approaches to ensuring data security in software systems in a cloud environment.

The aim of the work is to enhance the data security level in cloud applications by implementing various architectural approaches, including the Zero Trust model.

Development methods are based on technologies such as ASP.NET Core, .NET, Docker, AWS, Microsoft Azure.

As a result, various architectural approaches to data security were studied, and a comparative analysis of their effectiveness and impact on system performance was conducted. A prototype of a software system for handling patient data was developed, incorporating multi-layer data protection, multi-step authentication, and continuous activity monitoring. The system was subjected to penetration testing and other security tests to assess the level of protection provided by different architectural approaches. Based on the results, recommendations for improving data security in cloud applications were provided.

Умови публікації звіту: заява щодо самостійного виконання кваліфікаційної роботи та можливості її публікації в електронному архіві відкритого доступу ElArKhNURE.

Я, __Трибух Арсеній Олексійович_____
(прізвище, ім'я, по батькові)

студент групи __ІПЗм-22-1_____здобувач вищої освіти на другому (магістерському) рівні кафедра_____ програмної інженерії _____,
(повна назва кафедри)

заявляю: моя кваліфікаційна робота на тему __Дослідження архітектурних підходів до забезпечення приватності конфіденційних даних у хмарному середовищ_____,
(назва роботи)

що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElArKhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений (а) з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

Дата _____ Підпис _____

ЗМІСТ

ВСТУП

Сучасний світ інформаційних технологій став свідком стрімкого розвитку хмарних технологій, що стали важливим інфраструктурним компонентом для багатьох організацій та користувачів. Збільшення обсягів конфіденційної інформації, що зберігається та обробляється у хмарних сервісах, породжує актуальні питання щодо забезпечення приватності та безпеки даних.

Однією з ключових проблем, які виникають у контексті хмарних технологій, є необхідність забезпечення високого рівня приватності конфіденційних даних, які обробляються у віртуальних середовищах. Це стає особливо важливим у зв'язку з розповсюдженням різноманітних сервісів, що пропонуються через хмарні платформи, таких як зберігання даних, обчислення та аналіз. Забезпечення конфіденційності в умовах хмарних обчислень вимагає глибокого розуміння різних архітектурних підходів та їхнього впливу на рівень захисту особистих даних.

Спостереження за сучасними тенденціями використання хмарних технологій також вказують на те, що деякі компанії виявляють певні сумніви та облаштовують власні дороговартісні інфраструктури відмовляючись від хмарних сервісів. Це рішення зумовлене, переважно, питаннями приватності та безпеки даних, що є однією з найважливіших перешкод для широкого впровадження хмарних технологій. Певні компанії виражають занепокоєння стосовно того, як їхні конфіденційні дані можуть зберігатися та оброблятися сторонніми хмарними постачальниками. Це особливо актуально для сфер, де існують високі вимоги до захисту конфіденційної інформації, таких як фінансові установи, медичні організації та компанії, які працюють з особистими даними клієнтів.

Такі сумніви можуть бути пояснені невпевненістю в тому, наскільки ефективно хмарні сервіси забезпечують приватність та безпеку даних. Компанії можуть вагатися перед переходом до хмарних технологій через бажання зберегти контроль над власними даними та уникнути можливих ризиків порушення конфіденційності. Це також може бути пов'язане із строгими регуляторними

вимогами або внутрішніми політиками компаній, що визначають обраний підхід до обробки та зберігання конфіденційної інформації.

Отже, проблема приватності в хмарі є однією з ключових перешкод для широкого впровадження цих технологій в деяких галузях. Дослідження архітектурних підходів до забезпечення конфіденційності даних стає важливим в контексті зусиль щодо подолання цих обмежень та створення довірчого середовища для використання хмарних технологій у різних сферах бізнесу.

Мета даної кваліфікаційної роботи полягає у системному дослідженні та аналізі архітектурних підходів до забезпечення приватності конфіденційних даних у хмарних середовищах. Дослідження цього питання є актуальним, оскільки воно дозволяє визначити ефективні та надійні стратегії забезпечення конфіденційності даних в умовах постійного розвитку хмарних технологій.

В ході роботи буде розглянуто та проаналізовано аспекти різних архітектурних рішень, спрямованих на забезпечення безпеки та конфіденційності даних в хмарних середовищах, враховуючи сучасні виклики та тенденції у цій галузі. За результатами аналізу, буде обрано декілька підходів та створено прототип програмної системи із застосуванням їх застосуванням. Це дозволить провести аналіз впливу того чи іншого рішення чи підходу на безпеку даних та продуктивність системи. Також будуть враховані переваги та обмеження кожного підходу, що дозволить визначити оптимальні стратегії для різних сценаріїв використання хмарних технологій. Результатом роботи стануть рекомендації щодо можливості застосування чи поєднання тих чи інших підходів для забезпечення найкращого рівня безпеки даних та оптимальної продуктивності програмної системи

Отже, кваліфікаційна робота спрямована на висвітлення та розуміння суті архітектурних рішень, спрямованих на забезпечення конфіденційності даних у хмарних середовищах, що стане важливим внеском у вивчення та вдосконалення сучасних засобів захисту особистої інформації в хмарному середовищі.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Аналіз проблем зі зберіганням та обробкою конференційних даних при використанні хмарної інфраструктури

Розглянемо основні проблеми та виклики з якими стикаються користувачі та розробники програмних продуктів, що використовують хмарну інфраструктуру для розгортання систем.

Першою проблемою з якою стикаються користувачі хмарних провайдерів – це недостатній контроль над даними. Для переважної більшості компаній критично важливим є те, щоб до даних, що зберігаються, мали доступ лише визначені люди. Тому в них можуть виникати цілком логічні питання: “хто ще має доступ до даних?”, “яким чином забезпечується безпека?”, “де дані зберігаються фізично?” і так далі. Проблеми можуть виникати в тому випадку, якщо адміністратори не правильно налаштовують права доступу до ресурсів та об'єктів в хмарному середовищі. Це може призвести до неправомірного доступу до конфіденційної інформації. Крім цього є ймовірність втратити дані через нестабільну роботу самого хмарного провайдера, або недостатніх заходів із захисту цих даних на стороні провайдера.

Як показує статистичне дослідження Verizon's Data Breach Investigations Report 2021, що було проведено в 2021 році, 61% усіх порушень безпеки даних виникають через неправильну конфігурацію прав доступу, 22% – через нестабільність роботи інфраструктури провайдера.

Якщо з першою причиною можна боротися за допомогою постійних аудитів та моніторингу системи на потенційні безпекові вразливості, то для боротьби з другою – дій користувачів буде недостатньо, адже інфраструктура провайдера є зоною відповідальності самого провайдера. Тому актуальним є питання, як розробники зможуть власними силами (архітектурними, або технічними рішеннями) додатково забезпечити приватність та мінімізувати можливий вплив вразливостей в інфраструктурі хмарного провайдера.

Іншою складністю, з якою можуть стикнутись компанії, при використанні хмари для розгортання власних програмних продуктів є необхідність відповідати певним регуляторним та законодавчим нормам щодо обробки персональних даних користувачів. Ці додаткові вимоги створюють неабиякий тиск на компанії, яким доводиться ретельно вивчати відповідність хмарних провайдерів цим вимогам. Це часто призводить до того, що компанії змушені відмовлятися від використання хмарних рішень, надаючи перевагу власним дата центрам. Що в свою чергу значно підвищую вартість підтримки програмного продукту і таким чином зменшує потенційний прибуток.

Прикладами регуляторних документів є GDPR (General Data Protection Regulation) та HIPAA (Health Insurance Portability and Accountability Act) [1].

GBPR – це е регуляторний норматив, що визначає правила збору, обробки та захисту персональних даних громадян Європейського Союзу (ЄС). Введений в дію з 25 травня 2018 року, GDPR надає громадянам більше контролю над їхніми особистими даними та встановлює обов'язки для організацій, які обробляють ці дані. Ось деякі принципи з цього документу:

- легітимність, справедливість, прозорість: організації повинні обробляти дані чесно та прозоро, з чітким визначенням цілей та правовою підставою;
- обмеження обробки: обробка даних повинна обмежуватися тільки тими даними, які є необхідними для визначених цілей;
- точність: дані повинні бути точними та обновлюваними при необхідності;
- обмеження зберігання: дані мають зберігатися тільки протягом необхідного періоду для досягнення визначених цілей;
- право на доступ: суб'єкти даних мають право запитувати та отримувати доступ до своїх персональних даних.
- право на виправлення: суб'єкти даних мають право виправляти неточності у своїх даних;

- право на видалення: суб'єкти даних мають право вимагати видалення своїх даних у певних обставинах.

В свою чергу HIPAA – це законодавчий акт США, призначений для захисту приватності та безпеки медичної інформації пацієнтів. Введений в дію в 1996 році, HIPAA встановлює стандарти для обробки та зберігання медичної інформації та надає пацієнтам певні права щодо їхніх медичних записів[2].

HIPAA встановлює стандарти та правила для електронного обміну медичною інформацією:

- приватність та безпека інформації: закон накладає обов'язки на організації з охорони здоров'я щодо захисту приватності та безпеки медичної інформації;
- правила конфіденційності: визначають обмеження доступу до медичної інформації та вимоги щодо її конфіденційності;
- правила безпеки: встановлюють технічні та адміністративні заходи для захисту медичної інформації в електронній формі;
- право на доступ: пацієнти мають право отримувати копії своїх медичних записів та іншої інформації, пов'язаної з їхнім здоров'ям;
- право на контроль: пацієнти мають право контролювати, як їхні медичні дані використовуються та розголошуються [3].

Є й інші законодавчі акти та документи, які регулюють роботу з персональними даними, але ці два є найбільш розповсюдженими, тому для дослідження будемо рівнятися саме на них.

Ще однією проблемою, яка може виникнути при роботі з даними в хмарі – це проблема доступу до даних. Хмарні додатки можуть бути піддані атакам типу відмова в обслуговуванні (DoS) та розподілена відмова в обслуговуванні (DDoS) [4]. Такі атаки спрямовані на те, щоб перевантажити сервери та зробити сервіси недоступними для користувачів. Це може призвести до значних фінансових втрат та втрати довіри з боку користувачів.

Для боротьби з даною проблемою необхідно використовувати резервне копіювання, планування міграцій даних і т.д, а також спеціальні механізми, що запобігають DDos атакам.

Також, слід враховувати те, що хмарні провайдери часто використовують спільну інфраструктуру для обслуговування багатьох користувачів. Це може створювати додаткові ризики, що пов’язані з ізоляцією ресурсів і даних між різними користувачами. При неправильний конфігурації системи, або недостатньому захисту даних з боку провайдера можуть призвести до витоку даних.

Крім технічних векторів атаки на програмні системи присутні також і ризики, що пов’язані з людським фактором. Інсайдери, такі як співробітники або партнери з доступом до хмарних систем, можуть стати джерелом загроз. Вони можуть навмисно або випадково зловживати своїм доступом для викрадення або пошкодження даних, що може мати катастрофічні наслідки для організації. Більшість великих компаній проводять для своїх співробітників спеціальні тренінги, щодо захисту від соціальної інженерії, проте розробники програмних систем також мають вживати дій, щодо захисту від можливої втрати паролю користувачем, чи несанкціонованого доступу [5]. Ресурси мають бути максимально ізольовані один від одного, щоб несанкціонований доступ до певної частини системи не призводив до компрометації усієї системи.

1.2 Огляд існуючих способів забезпечення безпеки даних

Розглядаючи та аналізуючи підходи для забезпечення безпеки даних в хмарі, я прийшов до висновку, що їх можна розділити на декілька логічних груп. Кожна група відповідає за безпеку даних на своєму рівні. Нижче будуть перелічені ці групи:

- регуляція та відповідність законодавчим нормам;
- ідентифікація, аутентифікація та авторизація користувачів;
- приватність та конфіденційність даних;

– безпечна архітектура програмних систем.

Необхідно розуміти, що для того, щоб гарантувати безпеку даних, програмна система має відповідати нормам якості на всіх рівнях одночасно. Наприклад, як би якісно і безпечно не шифрувалися дані, що зберігаються в хмарній базі-даних, якщо хмарний провайдер має проблеми з аутентифікацією і зловмисник зможе отримати доступ до ключів шифрування, то така система є вразливою, не дивлячись на те, що певні її частини начебто є захищеними.

Також, хочеться згадати модель CIA Triad [6], яка є основоположною концепцією в області інформаційної безпеки. Вона складається з трьох основних компонентів: конфіденційності (Confidentiality), цілісності (Integrity) та доступності (Availability). Разом ці три елементи забезпечують всебічний підхід до захисту даних та інформаційних систем. На рисунку 1 зображено графічне представлення CIA Triad.

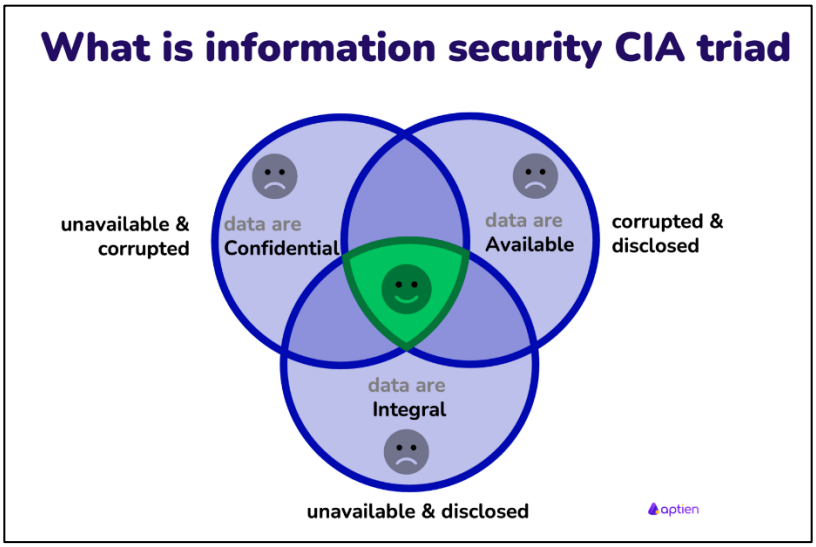

Рисунок 1.1 – Графічне представлення моделі CIA Triad (за даними [6])

Конфіденційність полягає в забезпеченні того, що інформація доступна тільки тим особам, які мають на це відповідні права. Це досягається за допомогою різних методів, таких як шифрування даних, контроль доступу та автентифікація користувачів.

Цілісність означає забезпечення того, що дані залишаються точними, повними та захищеними від несанкціонованих змін або руйнування. Це важливо для збереження достовірності та надійності інформації.

Доступність полягає в забезпеченні того, що інформаційні системи та дані доступні для авторизованих користувачів у будь-який час. Це включає забезпечення безперебійного функціонування систем навіть у випадку атак чи відмов обладнання.

Усі заходи безпеки зазвичай направленні на підтримку одного чи декількох аспектів з трьох вище перелічених. Тому в кваліфікаційній роботі не раз будуть зустрічатися відсилки до цієї концепції.

1.2.1 Регуляція і відповідність законодавчим нормам

Розглянемо деякі аспекти законодавчої регуляції щодо роботи з даними з якими можуть зіштовхнутися розробники програмних систем, а також загальні підходи до роботи з цими вимогами.

«Громадянство» даних (Data Citizenship).

Різні правові та нормативні вимоги та стандарти в різних географічних регіонах можуть вимагати фізичного зберігання певних типів даних у визначеній країні/юрисдикції. Наприклад, GDPR ЄС розрізняє зберігання й передачу даних у межах ЄС та експорт і зберігання даних за межі ЄС.

Хмарні провайдери пропонують свої послуги з тегами розташування. Під час створення екземпляра послуги користувач хмари може вибрати географічне розташування, яке визначається регіональним позначенням (наприклад, ЄС-Захід). Хоча хмарні постачальники зазвичай не афішують точне фізичне розташування своїх центрів обробки даних, вони надають гарантії того, що визначення географічного розташування підпадає під певну юридичну юрисдикцію. Однак географічні позначення не поширюються на всі хмарні служби; Великі хмарні середовища залишаються принаймні частково незалежними від розташування,

особливо для служб, які потребують розосередженої інфраструктури для забезпечення функціональності, таких як DNS або брандмауери веб-додатків.

Криптографічне видалення даних (Cryptographic Erasure) [7].

У хмарних середовищах, включно з розгорнутими кінцевими пристроями, дані часто копіюються та передаються між великою кількістю фізичних пристроїв, іноді географічно розосереджених. Це робить гарантоване безпечне видалення даних у традиційному розумінні важким, якщо не неможливим.

Шифрування даних у стані спокою зводить проблему керування видаленням усього набору даних до проблеми керування часом життя криптографічного ключа. Оскільки криптографічні ключі, які використовуються для шифрування в стані спокою, малі, ними набагато легше керувати, ніж потенційно величезні набори даних, і їх можна зберігати в контрольованому сховищі (наприклад, HSM). Тоді криптографічне видалення означає безпечне знищення ключових даних. За умови, що ключі не були скомпрометовані протягом усього терміну служби та використовувалися криптографічні алгоритми безпеки, криптографічне видалення гарантує нерозбірливість зашифрованого набору даних, аж до гарантій безпеки, наданих алгоритмом шифрування.

Модель спільної відповідальності (Shared Responsibility Model) [8].

Однією з переваг хмарних технологій є зниження загальної вартості володіння, а також відповідальності за функціонування хмарної інфраструктури. Придбання та керування власним центром обробки даних для запуску додатків пов’язане з багатьма прихованими витратами, нормативними вимогами та ризиками, які можуть стати значним тягарем для організації.

На початку початкових пропозицій хмарних послуг постачальники хмарних послуг пропонували лише інфраструктуру як послугу (IaaS), тобто обчислювальні засоби, де клієнти хмари могли встановлювати та запускати віртуальні машини. Відповідальність за безвідмовну роботу, надійність, доступність і безпеку інфраструктури лежить на хмарному провайдері, тоді як орендарі беруть на себе відповідальність за вибір, установку, обслуговування та роботу віртуальних машин і встановлених на них програм. У міру того як хмарні пропозиції переросли в більш

складну область платформи як послуги, відповідальність за все більшу функціональність взяли на себе постачальники хмарних послуг.

Передача даних (Complient Data Transfer).

Сучасні програми SaaS часто складаються з кількох API. Наприклад, онлайн-магазин може зосередити свою власну логіку програми на конкретному каталозі продуктів, але потенційно передасть стандартні функції, такі як вхід користувача, вхід, сповіщення електронною поштою, виставлення рахунків тощо стороннім постачальникам. Ці функції сторонніх розробників часто доступні через API і є частиною бізнес-пропозицій в економіці API.

Різні закони та нормативні акти включили концепцію сумісної передачі даних у свій корпус. Наприклад, GDPR ЄС передбачає можливість відповідної передачі даних обробникам даних третіх країн за допомогою договірних «типових положень».

Час зберігання даних (Data Retention).

Закони про конфіденційність у різних країнах накладають обмеження на здатність організацій зберігати деякі типи особистої інформації, і кожна країна має свій власний період зберігання. Таким чином, контроль збереження даних має бути адаптованим до юрисдикції, під якою працює хмарний сервіс. Крім того, у хмарних сценаріях управління зберіганням даних (громадянство даних) може відрізнятися від керування користувачами даних; напр. Азійські клієнти, які використовують місце зберігання даних у Європі.

Щоб уникнути потенційного порушення законодавства, слід використовувати автоматичний модуль (або інструмент) для контролю періоду зберігання даних. Коли законодавчо дозволений період зберігання закінчиться, дані мають бути остаточно стерті зі сховища. Автоматизація цього процесу полегшує керування даними, а також додатками [9].

Захист даних від випадкового видалення.

Деякі типи даних, відповідно до вимог законодавства, повинні зберігатися протягом певного періоду часу з метою розслідування чи дослідження. Наприклад, дані, згенеровані загальнодоступною відеокамерою, потрібно деякий час зберігати

для цілей розслідування. У хмарному контексті дані розділені логічно, але не фізично. Якщо зловмисник може отримати дані та навмисно їх знищити, програма або постачальник хмарної платформи надійні в цьому. План відновлення даних, розроблений для такої події, має бути на місці, щоб вирішити проблему втрати даних.

Рішення полягає в тому, щоб спроектувати систему таким чином, щоб представлення цифрових даних залишалося навіть після того, як були зроблені спроби видалити або стерти їх. Це стає можливим завдяки використанню реплікації/надмірності даних у фізично розподіленій хмарній системі.

1.2.2 Ідентифікація, аутентифікація та авторизація користувачів

У цьому розділі розглянемо якими способами та методами забезпечується безпечна аутентифікація користувачів в хмарні системи.

Мультифакторна аутентифікація (Multi-Factor Authentication) [10].

Автентифікація людей машинами є проблемою збалансування зручності використання та безпеки. Комбінація традиційних трьох факторів: те, що користувач знає (секретний пароль), те, що користувач має (фізична власність) і те, що є унікальною рисою користувача (біометрія), забезпечує високий рівень безпеки, кожен з яких накладає різний тягар на частина користувача. Паролі були основним фактором автентифікації протягом історії обчислювальної техніки, і існує велика кількість знань, які вказують на недоліки в обробці паролів користувачами. Фізичні токени часто використовуються як другий фактор автентифікації.

Хоча надання всіх трьох факторів автентифікації одночасно залишається найбезпечнішим варіантом, цей рівень безпеки часто не потрібен у типових сценаріях хмарних програм. Щоб пом’якшити вразливість, пов’язану з паролями, багатофакторні системи часто включають рівні автентифікації на основі сценаріїв доступу, рівня чутливості та пов’язаного з ними ризику операцій, які бажає виконати користувач; напр. у банківській програмі користувач може пройти

автентифікацію за допомогою відбитка пальця на своєму мобільному пристрої (2 фактори), щоб отримати доступ до своїх рахунків для перегляду, але великий грошовий переказ може вимагати додаткового введення пароля або PIN-коду.

Аутентифікація на рівні запиту (Per-request Authentication).

У нинішньому хмарному середовищі немає постійного контролю над діяльністю користувачів після того, як користувач пройшов автентифікацію. Якщо зловмиснику вдасться зламати обліковий запис, він може робити все, що завгодно, з ресурсом користувача та системи. Контроль дій користувача під час сеансу використання важливий у деяких випадках використання (наприклад, розумний дім, охорона здоров'я), щоб запобігти або мінімізувати шкоду, яка може виникнути в результаті злому облікового запису. Завдяки безперервному контролю система може вчасно реагувати на незвичайні дії, які виконує або виконує користувач.

Рішення полягає в розробці інструментів інтелектуального контролю використання, які відстежують діяльність користувача від початку до кінця сеансу використання. Інструменти мають працювати у фоновому режимі та бути достатньо інтелектуальними, щоб виявляти будь-які ненормальні дії та запобігати подальшому пошкодженню користувача у разі виявлення аномальних дій.

Двостороння аутентифікація.

У хмарному середовищі кілька фізичних і логічних компонентів з'єднуються та обмінюються інформацією. Без належної автентифікації між сторонами, що спілкуються, можливі атаки типу "людина посередині" (man in the middle). Використання двосторонньої автентифікації, щоб дозволити обом суб'єктам у каналі зв'язку автентифікувати один одного.

Використання стороннього серверу аутентифікації.

Управління ідентифікацією користувачів часто є обтяжливим завданням, і в додатках SaaS воно часто забирає багато часу, якщо його виконувати правильно.

Повторне використання існуючих функцій входу та входу користувачів, розроблених і підтримуваних третіми сторонами, є ефективним способом передачі завдань автентифікації третій стороні. Хоча технічні рішення, які використовують треті сторони, часто є найсучаснішими, такий аутсорсинг несе неминучі ризики

щодо захисту конфіденційності користувачів, особливо коли об’єднані організації є соціальними мережами.

1.2.3 Приватність та конфіденційність даних

У цьому підрозділі розглянемо деякі способи, якими можна забезпечити приватність та конфіденційність даних користувачів при розробці хмарних систем.

Обчислення над зашифрованими даними (Computation on Encrypted Data) [11].

Досить розповсюдженим випадком є необхідність виконувати певні обчислення чи операції над даними в хмарі, коли самі дані зберігаються на on-premises серверах. Наприклад за допомогою безсерверних рішень (таких як AWS Lambda). Проте, якщо дані є надзвичайно конференційними, в силу вступає вимога забезпечити неможливість розкриття даних в процесі обробки. Хмарні сервіси забезпечують привабливу еластичну модель обчислень; однак, щоб використовувати хмару крім простого зберігання зашифрованих даних, ключі повинні бути доступні постачальнику хмари, щоб розшифрувати дані перед обчисленням на них. У моделях загроз, де хмарі майже повністю довіряють, це створює проблему безпеки.

Рішенням буде використання повністю гомоморфної схеми, яка дозволяла складну обробку, навіть якщо дані були зашифровані, і користувачі не могли їх бачити.

Анонімізація даних (Data Anonymisation).

Хмарні служби часто використовуються для обробки великих наборів даних, які потенційно містять первинні чи вторинні приватні дані, або такі дані можуть бути отримані шляхом кореляції різних наборів даних.

Ідентичність власника даних має бути вилучена із записів таким чином, щоб власника даних неможливо було ідентифікувати прямо чи опосередковано з цих анонімних даних.

Контроль використання даних (Processing Purpose Control).

Під час створення хмарної програми для зберігання та обробки особистої та конфіденційної інформації (наприклад, системи охорони здоров’я) виникає багато проблем. Проблеми варіюються від безпеки до правових аспектів; Одна зі складних проблем, яку потрібно вирішити, полягає в тому, щоб дані, якими обмінюються різні зацікавлені сторони в мережі, використовувалися відповідно до їх визначеної мети. Відповідно до закону (наприклад, GDPR), обробник даних несе відповідальність за неправомірне використання особистої інформації та несе юридичну відповідальність за те, щоб дані, оброблені в його системі або в іншій системі, з якою вони надають доступ, використовувалися відповідно до закону, заявленої мети та згоди користувача.

Для контролю використання даних потрібні надійні інструменти контролю використання даних. Інструмент контролю використання дозволяє користувачеві не тільки контролювати та контролювати використання даних, але також відстежувати та перевіряти їх використання.

1.3 Огляд існуючих архітектурних підходів для забезпечення безпеки даних

У попередньому розділі було розглянуто деякі методи, які можуть використовуватись для захисту конференційних даних в хмарному середовищі. Більшість розглянутих методів зазвичай застосовуються на рівні хмарного провайдера чи адміністратора системи. В цьому розділі увагу буде приділено захисту даних з точки зору програмної інженерії та розробників. Сфокусуємось на архітектурних методах та прийомах. Буде розглянуто найбільш популярні моделі та стратегії та проаналізуємо їх використання хмарними провайдерами. Також, будуть зроблені висновки щодо можливості використання певних підходів та архітектурних методів в дослідженні.

Перший стратегічний підхід до розробки ПЗ, який буде розглянуто має назву «Zero Trust» [12] («Не довіряй нікому»). Це стратегічний підхід до безпеки інформації, який ґрунтується на принципі, що треба довіряти жодній особі чи системі, навіть якщо вони раніше розглядалися як довірені. Кожен користувач і

ресурс розглядаються як потенційний ризик, і вимагається аутентифікація та авторизація для кожної операції. Ця концепція випливає з реалій сучасного інтернет-простору, де традиційні засоби захисту втрачають актуальність через розподіленість робочих середовищ.

Цей підхід має декілька основних принципів, які будуть перелічені нижче:

- кожен запит до будь-яких ресурсів повинен бути аутентифікований та авторизований незалежно від того, чи є цей запит внутрішнім (між сервісами однієї системи), чи зовнішнім (той, що прийшов із зовнішньої мережі);
- шифрування усіх даних при передачі між компонентами системи: це означає, що всі дані, які передаються по мережі мають бути зашифровані, навіть якщо обмін відбувається в локальній і здавалося б захищеній мережі;
- мікросегментація – принцип розділення мережі на мікросегменти з обмеженим доступом для мінімізації так званого «бокового руху» зловмисників;
- нікому не надається довіра автоматично, кожен запит та дія має бути оцінений окремо на предмет ризиків.

Впровадження даного принципу дає можливість забезпечити найвищий рівень безпеки системи. Він може стосуватися як самої системи, так і даних, які вона зберігає та оперує. Стратегія здатна забезпечити безпеку в умовах розподіленості ресурсів, проте вона має певні недоліки. При застосуванні підходу значно збільшується навантаження на систему аутентифікації, яка може стати точкою відмови у разі виникнення проблем із нею. Тому треба обов’язково враховувати це при запроваджені даного підходу та забезпечити бездоганну доступність та відмовостійкість для сервісів, що беруть участь в аутентифікації.

Наступним підходом, що буде розглянуто є «Шифрування на рівні даних». Він передбачає, що дані шифруються на рівні, де вони зберігаються або пересилаються. Цей підхід призначений для мінімізації ризиків, пов'язаних з втратою або несанкціонованим доступом до конфіденційної інформації, навіть у випадку фізичного доступу до зберігаючих пристроїв чи систем. Дані шифруються

з використанням алгоритмів шифрування, перетворюючи їх у незрозумілий формат, який вимагає ключа для дешифрування.

При цьому самі ключі шифрування відокремлюються від самих даних і зберігаються в безпечному місці. Вони використовуються тільки для дешифрування даних, забезпечуючи контроль доступу. Ключі шифрування можуть генеруватися для кожного окремого набору даних або навіть для кожного запису, збільшуючи важкодоступність для несанкціонованого доступу.

Таким чином, даний підхід забезпечую безпеку даних при зберіганні, адже зловмисник не зможе прочитати дані, навіть, якщо він отримує доступ до сховища. Проте дані все ще можуть бути вразливі при передачі, якщо будуть передаватися у незашифрованому вигляді. Графічна схема такої вразливості буде представлена на рисунку 1.2.

Тому є доцільним комбінувати даний підхід з шифруванням даних на рівні передачі, щоб забезпечити те, що вони не будуть перехоплені.

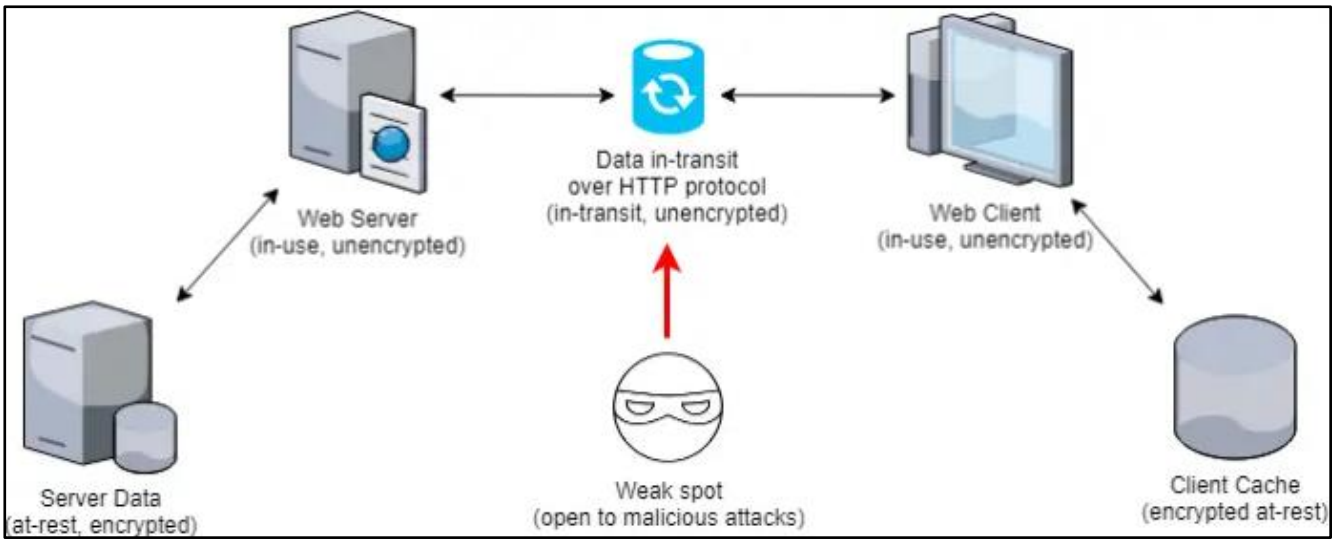

Рисунок 2.1 – Графічна схема можливої вразливості підходу «Шифрування на рівні даних» (за даними [13])

Наступний підхід, який хочеться розглянути є досить очевидним, але в цей же час і ефективним, а саме використання мікросервісної архітектури. Заключається він в тому, щоб розбити систему на окремі автономні мікросервіси, які комунікують між собою. Це дозволить організувати окрему систему безпеки

для кожного з мікросервісів, що дозволить використовувати найбільш підходящі засоби та методи для кожного конкретного випадку. Наприклад, мікросервіси, які оперують конфіденційними будуть захищені найкраще, при цьому для сервісів де такий захист не потрібен, він не буде іплементований, що дозволить уникнути використання зайвих обчислювальних потужностей [15].

В цей же час, мікросервісна архітектура є складнішою в розробці та підтримці, адже вимагатиме додаткового управління безпекою та іншими параметрами на рівні кожного окремого мікросервісу.

Ще один цікавий метод, що можна використовувати для забезпечення приватності даних на рівні бізнес-організацій (клієнтів хмарних провайдерів), або навіть кінцевих користувачів системою має назву «Bring Your Own Key», або «BYOK» («Принеси свій власний ключ»). Підхід заключається в тому, що дані шифруються за допомогою ключа, що був створений та контролюється користувачем. З цього випливає, що тільки той, хто дійсно має право доступу до даних отримає його, адже тільки він матиме ключ, необхідний для розшифрування.

Такий підхід до управління даних може використовуватись організаціями, які підпадають під дію регулюючих документів і може стати вирішенням даної проблеми.

2 ПЛАНУВАННЯ ДОСЛІДЖЕНЬ ТА ЕКСПЕРИМЕНТІВ

2.1 Визначення мети дослідження

Мета цього дослідження полягає в ретельному вивченні та аналізі різних архітектурних підходів до забезпечення приватності конфіденційних даних у хмарному середовищі. Основна мета полягає в ідентифікації ефективних стратегій та методів, що забезпечують високий рівень захисту особистої інформації в хмарних обчисленнях.

Конкретні цілі дослідження включають:

- аналіз існуючих архітектурних рішень: дослідження різноманітних архітектурних моделей, що застосовуються в хмарних сервісах для забезпечення конфіденційності даних;
- оцінка рівня безпеки та приватності: визначення ступеня ефективності та ризиків, пов'язаних з кожним архітектурним підходом в контексті забезпечення приватності даних.
- розробка експериментального плану: встановлення конкретних критеріїв, методів та технік, які будуть використовуватися для оцінки ефективності кожного архітектурного рішення;
- вивчення впливу на продуктивність: аналіз впливу застосування різних архітектурних підходів на продуктивність системи та роботу хмарних сервісів;
- розробка рекомендацій: визначення найбільш перспективних та безпечних архітектурних підходів і розробка рекомендацій для практичного впровадження у реальних умовах;

Це дослідження має на меті сприяти розвитку практичних рекомендацій для бізнес-середовищ, які планують або вже використовують хмарні рішення, забезпечуючи найвищий ступінь захисту конфіденційних даних своїх користувачів.

Для проведення дослідження будемо використовувати два різних хмарних провайдера. Які саме буде визначено в наступному розділі. Спочатку буде

побудовано прототип хмарної системи без використання додаткових архітектурних засобів для захисту даних. Це буде базова еталонна версія системи. Систему буде розгорнуто в інфраструктурі двох різних хмарних провайдерів. Буде виконано заміри продуктивності та безпечності даних для неї, щоб згодом ми могли оцінити вплив того чи іншого архітектурного рішення на результати.

2.2 Обґрунтування вибору cloud-провайдерів для проведення дослідження

Перше рішення, яке треба прийняти для початку даного дослідження - на основі яких cloud-провайдерів його проводити. На сьогоднішній день існує кілька популярних хмарних рішень. Наприклад AWS, Microsoft Azure, тощо. Жодне з цих рішень, на даний момент, не може забезпечити повну безпеку та приватність даних. Проте кожне з них, так чи інакше має певні інструменти для цього. Тому для свого дослідження я ухвалив рішення взяти один (або два) cloud-провайдери, які найкраще підходять за певними критеріями (див. далі), тому на даний момент, моя задача буде визначити найбільш підходящі зі списку найпопулярніших в світі провайдерів. Визначати це будемо шляхом вирішення задачі багатокритеріального вибору. Тож нижче буде приведена постановка задачі багатокритеріального вибору.

2.2.1 Постановка задачі багатокритеріального вибору провайдерів

Задача багатокритеріального вибору буде сформульована наступним чином: «Вибрати 1-2 cloud-провайдера, які найкраще підходять для проведення подальшого дослідження та експериментів зі списку: Amazon Web Services, Microsoft Azure, Google Cloud, IBM Cloud, Alibaba Cloud) за наступними критеріями: рівень безпеки та захисту даних, відмовостійкість, географічна розгортка дата-центрів та доступність сервісів у різних регіонах, вартість використання та доступність тарифів, популярність»

2.2.2 Опис критеріїв

Тепер розглянемо кожен з критеріїв окремо для обґрунтування їх вибору.

Безпека та конфіденційність.

У вимірах обробки конфіденційної інформації безпека та конфіденційність стають пріоритетом. Захист від несанкціонованого доступу, шифрування даних та відповідність стандартам безпеки гарантують, що конфіденційна інформація залишається захищеною.

Вартість та економічна доцільність.

З урахуванням фінансових обмежень та стратегій оптимізації бізнесу важливо визначити витрати на користування хмарними послугами. Ефективне використання ресурсів та вартість послуг впливають на економічну доцільність та прийняття рішень.

Географічна розгортка.

Географічна розгортка центрів обробки даних визначає доступність та швидкість обслуговування клієнтів. Оптимальне розташування може забезпечити низький пінг та ефективність передачі даних у реальному часі.

Відмовостійкість.

У сучасному хмарному середовищі, відмовостійкість грає ключову роль у забезпеченні надійного обчислення. Адже downtimи можуть призвести до втрати великих коштів.

Популярність.

Популярність хмарної платформи є ключовим критерієм, оскільки вона відображає рівень довіри та прийняття користувачами та бізнесами. Висока популярність може свідчити про добре побудовану інфраструктуру, надійність послуг та задоволення користувачів. Популярні платформи часто мають активні та розвинені спільноти, що сприяє обміну досвідом та доступом до широкого спектру рішень.

Ці критерії визначають основні аспекти хмарного обчислення, які можуть впливати на вибір платформи в контексті дослідження архітектурних підходів до забезпечення приватності конфіденційних даних у хмарному середовищі.

Оптимальний вибір платформи повинен враховувати баланс між цими критеріями.

2.2.3 Опис шкал за обраними критеріями

Розглянемо за якими шкалами будемо оцінювати кожний із критеріїв.

Географічна розгортка дата-центрів та доступність сервісів у різних регіонах.

Візьмемо основні регіони світу: Північна Америка, Європа, Азія + Тихоокеанський регіон, Південна Америка, Африка.

Тепер оцінимо присутність cloud-провайдера в кожному регіоні, по шкалі від 0 до 1, де: 1 – це присутній, 0.5 – присутній частково (починає заходити в регіон), 0 – не присутній.

При цьому, слід зауважити, що кількість технологічних компаній (потенціальних користувачів провайдерів) не рівномірно розподілена по світу. Тому має сенс використати вагові коофіцієнти при розрахунку. Статистика показує, що 40% ринку приходиться на Північну Америку, 21% на Європу та 19% на Азію, по 10% на Африку та Південну Америку і .

Таким чином, кінцева шкала оцінює присутність на світовому ринку від 0 до 1, до 0 – це зовсім не присутній, 1 – присутній усюди.

Покажемо детальний розрахунок значення на прикладі Alibaba Cloud.

Він дуже сильно розповсюджений в Азійському регіоні (1), частково знаходиться в Європі (0.5) та США (0.5), в Африці (1) та Південній Америці (1) не представлений. Тому:

Азія – 1* 0.19 = 0.19

Північна Америка – 0.5 * 0.4 = 0.2

Європа – 0.5 * 0.21 = 0.105

Африка – 0 * 0.1 = 0

Південна Америка – 0 * 0.1 = 0

$$G(\text{Alibaba}) = 0.19 + 0.2 + 0.105 + 0 + 0 = 0.495$$

Отримане значення буде занесено в векторну таблицю і потім порівняно з іншими провайдерами.

Аналогічним чином були розраховані значення для кожного з провайдерів.

$$G(\text{GSP}) = 0.5 * 0.19 + 1 * 0.4 + 1 * 0.21 + 0.5 * 0.1 + 0.5 * 0.1 = 0.805$$

$$G(\text{Azure}) = 1 * 0.19 + 1 * 0.4 + 1 * 0.21 + 0.5 * 0.1 + 0.5 * 0.1 = 0.9$$

$$G(\text{AWS}) = 1 * 0.19 + 1 * 0.4 + 1 * 0.21 + 0.5 * 0.1 + 0.5 * 0.1 = 0.9$$

$$G(\text{IBM}) = 0.5 * 0.19 + 1 * 0.4 + 1 * 0.21 + 0.5 * 0.1 + 0.5 * 0.1 = 0.805$$

Популярність

Популярність є досить суб'єктивною величиною. Задля оцінювання була взята статистика з декількох різних джерел, де в представників міжнародних IT компаній (компаній, що мають IT-продукт) запитували, чи користувалися вони в роботі певним cloud-провайдером. Шкала буде від 0 до 1, де: 100% (1) – усі опитані користувались провайдером, 0% (0) – жоден з опитаних не користувався провайдером.

Статистика показала, що абсолютним лідером є AWS, який використовували чи використовують 80% опитаних, на другому місці Microsoft Azure з 75%. Третє місце розділяють GCP та Alibaba, з 50%, і замикає список IBM Cloud зі значенням в 22%.

Для оцінювання популярності візьмемо шкалу від 0 до 1 і переведемо вказані вище відсотки в неї. Ці дані буде занесено в векторну таблицю.

Рівень безпеки та захисту даних

Для оцінювання рівню безпеки та даних, використаємо критерії, які мають бути присутні в захищеній системі, а саме:

- політики приватності – як провайдери регулюють та гарантують приватність даних користувачів;
- шифрування та захист даних – які методи шифрування використовуються для захисту конфіденційних даних та які інші заходи безпеки використовуються;

- способи зберігання та обробки даних – як провайдери зберігають та обробляють конфіденційні дані користувачів;
- управління дозволами та доступом – як вони здійснюють управління дозволами та доступом до конфіденційних даних;
- відповідність законодавству – як провайдери відповідають вимогам законодавства про захист персональних даних та інших відповідних правил і стандартів;
- механізми виявлення та реагування на вторгнення – як провайдери реагують на потенційні вторгнення або порушення конфіденційності даних.

За кожен з цих пунктів (при наявності) провайдер отримує 1 бал, за відсутності – 0. Якщо існують певні механізми, проте у користувачів виникають сумніви, то оцінка буде 0.5.

Таким чином шкала оцінювання за цим критерієм буде варіюватися від 0 до 6, де: 0 – жодних механізмів безпеки, 6 – усі необхідні механізми безпеки

Розрахунки за шкалою:

Нижче буде приведена таблиця 2.1 з оцінками цього критерію за вищезазначеною шкалою.

Таблиця 2.1 – Оцінка cloud-провайдерів (виконано самостійно)

	Політики	Зберігання та обробка	Управління доступами	Законо давство	Шифру вання	Реагування	Результат
AWS	1	0.5	1	1	1	1	5.5
Azure	1	1	1	1	1	0.5	5.5
GCP	1	0.5	1	1	1	0.5	5
IBM Cloud	1	1	1	1	1	1	6
Alibaba Cloud	1	0.5	1	0.5	1	0.5	4.5

Коментуючи таблицю вище, хочу зазначити, що всі провайдери, що розглядаються так чи інакше імплементують згадані механізми, тому оцінок 0 –

немає. Проте, оцінку 0.5 отримали провайдери, які мають інциденти з певних аспектів безпеки. Наприклад, AWS, хоча і вважається найпопулярнішим провайдером, викликає певні занепокоєння щодо приватності даних, які там зберігаються. Ту ж проблему має Alibaba Cloud, який окрім цього може мати певні проблеми з законодавством в деяких країнах (так як компанія з Китаю).

Таким чином результуючі дані з таблиці вище, будуть перенесені в векторну таблицю.

Відмовостійкість нижча 99.9% вважається поганою для cloud-провайдерів. Тому 99.9% ми візьмемо за 0. Інші три два знаки після коми і будуть нашим значенням. Наприклад значення 99.981 перетворюється на 0.81. Шкала буде варіюватися від 0 до 1.

Дані взяті з офіційних сайтів провайдерів показують що, AWS, Azure та IBM мають 99.999% uptime, GSP 99.98%, Alibaba – 99.95%. Тому перші три отримають значення 0.99, GCP – 0.8, Alibaba – 0.5.

Оцінка та порівняння вартості хмарних послуг є досить важкою задачею, так як вона дуже залежить від конкретних задач, сервісів та систем. Враховуючи, що для тієї задачі вибору, яку ми вирішуємо, вартість не є пріоритетним критерієм, оцінимо її за якісною шкалою згідно загальній статистиці. При цьому 0 – буде відповідати найдорожчому провайдеру, 4 – найдешевшому.

Згідно статистики найдешевшою альтернативою вважається, Alibaba тому отримую 4 бали. Найдорожчим варіантом є IBM Cloud, який отримує 0. Відповідно 1 отримує AWS, який також є досить дорогим, і 2 – GCP і Azure.

Переведемо значення із якісної шкали в кількісну. Фактично, значення залишаться тими самими, але потім будуть перетворені в шкалу від 0 до 1 при нормалізації.

2.2.4 Векторний опис альтернатив та нормалізація

В результаті розрахунку значень кожного критерію згідно з шкалами був сформований векторний опис альтернатив у вигляді таблиці (таблиця 2.2).

Таблиця 2.2 – Векторний опис альтернатив (виконано самостійно)

	Безпека	Відмовостійкість	Географічна розгортка	Популярність	Вартість
AWS	5.5	0.99	0.9	0.8	2
Azure	5.5	0.99	0.9	0.75	3
GCP	5	0.8	0.805	0.5	3
IBM Cloud	6	0.99	0.805	0.22	0
Alibaba Cloud	4.5	0.5	0.495	0.5	4

Проведемо нормалізацію значень в таблиці за принципом еталону. За еталон візьмемо найкраще значення за цим критерієм серед усіх альтернатив. В результаті усі значення таблицю будуть в проміжку від 0 до 1 включно. Використаємо формулу:

$$f = \frac{f_{\text{вим}}}{f_{\text{еталон}}}$$

Покажемо детальний розрахунок для критерія «Безпека». За еталон візьмемо значення 6, яке має IBM Cloud. Таким чином:

AWS: 5.5/6 = 0.916.

Azure: 5.5/6 = 0.916.

GCP: 5/6 = 0.833.

IBM: 6/6 = 1.

Alibaba: 4.5/6 = 0.75.

Аналогічним чином буде проведена нормалізація і для інших критеріїв.

Для відмовостійкості за еталон візьмемо значення 0.99.

Для географічної розгортки – 0.9.

Для популярності – 0.8.

Тоді нормалізована векторна таблиця набуває наступного вигляду (таблиця 2.3).

Таблиця 2.3 – Нормалізована векторна таблиця (виконано самостійно)

	Безпека	Відмовостійкість	Географічна розгортка	Популярність	Вартість
AWS	0.916	1	1	1	0.6
Azure	0.916	1	1	0.9375	0.75
GCP	0.833	0.808	0.894	0.625	0.75
IBM Cloud	1	1	0.894	0.275	0
Alibaba Cloud	0.75	0.505	0.55	0.625	1

В таблиці вище приведені нормалізовані оцінки по кожному критерію від 0 до 1. Тепер вони в одній шкалі і можуть використовуватися для подальших розрахунків з використанням згорткової моделі для визначення найкращих альтернатив.

2.3.5 Розрахунок альтернатив з використанням згорткової моделі

Проаналізувавши множину альтернатив, застосуємо принцип Парето на цій множині. Виявляється, що альтернатива Azure краща за альтернативу GCP за принципом Парето, адже Azure має краще показники по всім критеріям та не поступається у вартості. Тому альтернативу GCP можемо виключити з множини альтернатив та не розраховувати згорткову модель для неї.

Зауважимо, що для даної задачі, критерії мають не однакову вагу. Наприклад, безпека та відмовостійкість мають набагато вищий пріоритет, ніж ціна, тому що клієнти будуть готові платити більше, якщо рішення повністю задовольнить їхні потреби. Тому було ухвалене рішення використовувати лінійну адаптивну згортку з ваговими коефіцієнтами. Формула виглядає наступним чином:

$$Z^* = \max_{i=1,m} \sum_{j=1}^{n} \alpha_j \beta_j a_{ij}$$

Тепер необхідно визначити вагові коефіцієнти для критеріїв. Зробимо це пропорційним методом. Найбільшу вагу зі значенням 0.4 для нашої задачі матиме Безпека. Відмовостійкість також є дуже важливою, тому матиме коефіцієнт 0.2. Географічна розгортка і Популярність матимуть по 0.15, і Ціна матиме найнижчий пріоритет – 0.1. Занесемо ці дані в таблицю (таблиця 2.4).

Таблиця 2.4 – Нормалізована таблиця з коефіцієнтами (виконано самостійно)

	Безпека	Відмовостійкість	Географічна розгортка	Популярність	Вартість
AWS	0.916	1	1	1	0.6
Azure	0.916	1	1	0.9375	0.75
IBM Cloud	1	1	0.894	0.275	0
Alibaba Cloud	0.75	0.505	0.55	0.625	1
Ваговий коефіцієнт	0.4	0.2	0.15	0.15	0.1

Розрахуємо згорткову модель для кожної альтернативи.

AWS: 0.916*0.4 + 1*0.2 + 1*0.15 + 1* 0.15 + 0.6* 0.1 = 0.9264.

Azure: 0.916*0.4 + 1*0.2 + 1*0.15 + 0.9375* 0.15 + 0.75* 0.1 = 0.9320.

IBM Cloud: 1*0.4 + 1*0.2 + 0.894*0.15 + 0.275* 0.15 + 0* 0.1 = 0.7753.

Alibaba Cloud: 0.75*0.4 + 0.5 * 0.2 + 0.55 * 0.15 + 0.625 * 0.15 + 1 * 0.1 = 0.677.

З розрахунків за згорткової моделі бачимо, що найкращою альтернативою для нашої задачі вибору є Microsoft Azure. З мізерним відставанням (менше 1 відсотка) на другому місці альтернатива AWS і з великим відставанням усі інші альтернативи. Так як в умовах задачі допускалося обрати дві альтернативи, можна знехтувати різницею між першим і другим місцем і вважати альтернативи рівноцінними і для подальшого дослідження та експериментів обрати саме їх.

2.3 Аналіз обраних провайдерів, щодо функціональних можливостей та забезпечення безпеки даних

У цьому розділі ми проведемо аналіз двох провідних провайдерів хмарних послуг — Amazon Web Services (AWS) та Microsoft Azure — з точки зору їх функціональних можливостей та механізмів забезпечення безпеки даних. Це допоможе зрозуміти, які інструменти та методи безпеки вони надають, і як вони можуть бути використані в подальшому дослідженні для практичної реалізації прототипу програмної системи.

Нижче будуть перелічені найбільш популярні сервіси, що пропонує AWS [16]:

- EC2 (Elastic Compute Cloud): Налаштовувані віртуальні машини для обчислень;
- S3 (Simple Storage Service): Масштабоване хмарне сховище для будь-яких типів даних;
- RDS (Relational Database Service): Керовані реляційні бази даних, такі як MySQL, PostgreSQL та Oracle;
- Lambda: Безсерверні обчислення для автоматичного запуску коду у відповідь на події;
- VPC (Virtual Private Cloud): Ізольовані хмарні мережі для забезпечення безпеки та контролю.

Детальний огляд цих сервісів не є фокусом для цієї роботи, тому більше увагу приділимо сервісам, призначення яких забезпечити безпеку даних.

Наприклад Amazon Web Services пропонує службу AWS Key Management Service (KMS) для керування ключами шифрування та AWS Identity and Access Management (IAM) для управління доступом. Крім того, Amazon S3 може використовувати шифрування з використанням AWS Key Management Service (SSE-KMS) для захисту збережених даних.

AWS KMS - це служба для керування ключами шифрування, яка дозволяє створювати та керувати ключами для шифрування та розшифрування даних. Вона забезпечує прозорий доступ до шифрування для користувачів та додатків без

необхідності напряму взаємодіяти з ключами. AWS KMS використовується для захисту конфіденційних даних шляхом шифрування, і включає такі функції, як автоматичне обслуговування ключів та моніторинг їхнього використання.

AWS Identity and Access Management (IAM) – це служба для управління доступом, яка дозволяє створювати та управляти ідентифікаційними та правами доступу для користувачів AWS. Вона забезпечує гнучкі налаштування прав доступу до різних ресурсів та послуг AWS. IAM використовується для визначення, хто та в якому обсязі може взаємодіяти з різними ресурсами AWS. Це включає управління користувачами, групами та ролями, а також визначення політик доступу.

Amazon S3 та AWS Key Management Service (SSE-KMS) – це служба зберігання об'єктів, а SSE-KMS - це опція для шифрування даних, де ключі управляються за допомогою AWS KMS. При використанні SSE-KMS для Amazon S3, дані автоматично шифруються перед зберіганням, і лише за допомогою правильного ключа можуть бути розшифровані. Це додає додатковий рівень безпеки для зберігання конфіденційної інформації в S3.

В свою чергу Microsoft Azure [17] також надає свої сервісу для захисту даних користувачів. Microsoft Azure також пропонує широкий набір хмарних послуг, включаючи обчислювальні ресурси, зберігання даних, бази даних, аналітику, мережеві сервіси, інструменти розробки, AI та ML. Основні сервіси включають:

- Azure Virtual Machines: Налаштовувані віртуальні машини для обчислень;
- Azure Blob Storage: Масштабоване сховище для неструктурованих даних;
- Azure SQL Database: Керовані реляційні бази даних;
- Azure Functions: Безсерверні обчислення для запуску коду у відповідь на події;
- Azure Virtual Network (VNet): Ізольовані хмарні мережі для забезпечення безпеки та контролю;

Azure Information Protection (AIP) – це служба для класифікації та захисту даних в хмарному середовищі. Вона дозволяє користувачам присвоювати рівні конфіденційності та класифікації до даних, а також встановлювати політики безпеки для їхнього захисту. За допомогою AIP, користувачі можуть визначити, хто має доступ до певних даних, контролювати їхнє використання та відстежувати активність щодо конфіденційної інформації в режимі реального часу.

Azure Key Vault – це служба для безпечного управління ключами, секретами та сертифікатами. Вона дозволяє зберігати та керувати конфіденційною інформацією, такою як ключі шифрування, в централізованому та захищеному середовищі. Azure Key Vault використовується для забезпечення безпеки ключів шифрування, які використовуються в різних сервісах та додатках Azure. Він забезпечує централізований доступ та автоматизоване керування ключами.

Azure Active Directory (AAD) – це служба ідентифікації та управління доступом, яка забезпечує ідентифікацію користувачів та керування їхнім доступом до ресурсів. AAD використовується для централізованого управління ідентифікацією та авторизацією в хмарному середовищі Azure. Воно дозволяє налаштовувати рівні доступу, визначати права користувачів та забезпечувати безпеку вхідних даних в сервіси Azure.

Обидва провайдери пропонують схожий набір хмарних послуг, які можуть задовольнити потреби більшості організацій. AWS та Azure мають потужні можливості для обчислень, зберігання, управління базами даних, а також безсерверних обчислень. Основна відмінність полягає у підходах до інтеграції та екосистемах, які підтримують їх сервіси. AWS та Azure надають комплексні рішення для безпеки даних, включаючи управління ідентифікацією та доступом, управління ключами шифрування, захист від DDoS-атак, моніторинг та логування активності, а також управління політиками безпеки та відповідності. Обидва провайдери регулярно проходять сертифікації відповідності міжнародним стандартам безпеки, таким як ISO 27001, SOC 1/2/3 та інші.

2.4 Формування вимог до програмної системи

Основне призначення прототипу програмної системи – визначити вплив тих чи інших архітектурних рішень на безпеку даних та продуктивність. У якості прототипу – створимо систему аудиту пацієнтів. Вона буде імітувати роботу програмного забезпечення, що використовують лікарні для управління обліковими даними своїх пацієнтів, історією хвороб, візитами до лікаря тощо.

Прототип програмної системи повинен мати наступні функціональні можливості:

Аутентифікація та авторизація користувачів.

В системі будуть присутні три ролі – пацієнт, лікар, адміністратор лікарні (працівник реєстратури). В залежності від ролі, користувачі будуть отримувати доступ до різних частин системи і функціональності.

Реєстрація пацієнтів

Пацієнти реєструються самостійно через форму реєстрації. Вказують персональні дані, контактну інформацію, історію хвороб.

Управління обліковими записами лікарів.

Адміністратори лікарні можуть створювати видаляти та редагувати облікові записи лікарів.

Запис на прийом до лікаря.

Пацієнт може записатись на прийом до лікаря на певну дату і час. Також, адміністратор лікарні може самостійно записати пацієнта на прийом. Після прийому лікар може внести інформацію про діагноз пацієнта та рекомендації щдо лікування.

Перегляд історії записів для конкретного пацієнта.

Адміністратори лікарні можуть переглядати усю інформацію про пацієнтів. Лікар може отримувати інформацію тільки про пацієнтів, з якими у нього назначено записи.

Перегляд історії записів для конкретного лікаря.

Адміністратори можуть переглядати історію прийомів лікаря.

Запити для отримання різної статистики

Імпорт та експорт даних пацієнтів з сторонніх систем та в сторонні системи.

Програмна система повинна мати можливість інтеграції з іншими системами обліку пацієнтів за допомогою публічних API. Має бути реалізована можливість імпорту даних пацієнта із сторонніх систем, а також експорту вже змінених даних назад в сторонню систему.

З точки зору користувачів, така функціональність є досить важливою, так як вона зменшує необхідність постійно вводити свої дані. З точки зору дослідження, вона дозволить продемонструвати передачу конференційних даних у сторонні системи, а також провести певні атаки на них та спробувати захиститись.

Слід зазначити, що в рамках дослідницької роботи, буде реалізовано лише бекенд частину даної системи, включаючи рівень роботи з даними.

2.5 Суть експерименту

В ході підготовки до експерименту буде створено декілька версій однієї й тієї ж програмної системи. Кожна версія системи буде окремо розгортатись в інфраструктурі двох хмарних провайдерів – AWS та Microsoft Azure. Спершу система буде створена з застосування мінімальної кількості архітектурних підходів для захисту даних. Будуть зроблені заміри швидкості виконання основних запитів, використання оперативної пам’яті та навантаження на центральний процесор. Також буде зроблено тестування під навантаженням для найбільш ресурсозатратних запитів, щоб проаналізувати як система реагує на велику кількість запитів. Окрім цього, на систему будуть здійснені різні атаки (детальніше про атаки буде розказано в наступних розділах) з метою отримати несанкціонований доступ до конференційних даних. Буде оцінено здатність системи протистояти таким втручанням.

Після цього, в архітектуру системи будуть ітеративно вноситись зміни та застосовуватись різні архітектурні підходи для захисту даних. Змінену версію

системи буде знов опубліковано на інфраструктурі двох різних клауд провайдерів та повторено тести.

В результаті буде складена таблиця з усіма замірами та результатами, проведено аналіз даних, які вдасться зібрати та сформовано висновки та рекомендації щодо застосування тих чи інших архітектурних підходів для захисту даних.

В рамках дослідження будуть розроблені наступні версії системи:

- монолітна трирівнева архітектура без застосування спеціальних підходів для захисту даних. Розгорнута за допомогою докер контейнера;
- мікросервісна архітектура зі спільною базою даних;
- мікросервісна архітектура з фізичною ізоляцією даних для кожного окремого мікросервіса. Окремий сервіс для Аутентифікації. Без використання інших засобів захисту;
- застосування моделі Zero Trust для попередньої версії системи. Кожен запит до даних має бути аутентифікований. Навіть при комунікації між внутрішніми сервісами в системі;
- застосування шифрування даних в спокої і при передачі даних;
- поєднання Zero Trust та шифрування даних.

3 ПІДГОТОВКА ДО ЕКСПЕРИМЕНТУ. СТВОРЕННЯ ПРОТОТИПУ ПРОГРАМНОЇ СИСТЕМИ

3.1 Опис технологій, що використовуються

Для програмування бекенд частини програмної системи буде використовуватись платформа .NET 8 та мова програмування C#. Вибір мови програмування дозволить створити cloud-base систему, з використанням найбільш сучасних підходів та прийомів. Для створення API використаємо фреймворк ASP.NET.Core, що є найбільш сучасним рішенням для створення Web.Api проєктів на базі платформи .NET.

Для розгортання та публікації програмної системи буде використовуватись Docker [18] – платформ для автоматизації та розгортання додатків у контейнерах. Використання Docker дозволить розгортати та запускати додатки в будь-якому середовищі та легко переміщувати їх між різними серверами, операційними системами, та хмарними провайдерами, що в свою чергу дасть додаткову гнучкість при розробці. Окрім цього, використання Docker позбавить необхідності кожен раз налаштовувати сервери для розгортання додатків та встановлювати необхідне програмне забезпечення. Адже docker-контейнер всередині себе вже має усі необхідні залежності для запуску додатку [19].

Для зберігання даних системи буде використовуватися дві кардинально різних системи управління базами даних – реляційна та не реляційна. Реляційна база буде основною. Там зберігатиметься більшість даних системи, а саме інформація про пацієнтів, лікарів, прийоми тощо. У нереляційній базі зберігатимуться проміжні логи імпорту даних із сторонніх систем. Таке рішення було прийнято для того, щоб при дослідженні в нас було більше опцій для пошуку потенційних вразливостей та методів захисту від них.

У якості реляційної бази даних використовуватиметься MSSQL Server. Ця СУБД є однією з найпопулярніших в сфері програмної інженерії, а також підтримуються більшістю хмарних провайдерів. Схема бази даних буде розроблена в наступному підрозділі.

У якості нереляційної БД використовуватимемо MongoDB. Ця база даних є найбільш популярною серед нереляційних рішень. MongoDB пропонує гнучкість у моделюванні даних завдяки використанню документів у форматі BSON, що дозволяє зберігати дані різних структур без необхідності суворої схематики. Вона підтримує горизонтальне масштабування через шардінг, що ефективно обробляє великі обсяги даних та високі навантаження [20]. MongoDB забезпечує високу продуктивність, швидкий доступ до даних та швидкість запису. Також вона має багаті можливості запитів та індексації, а вбудована підтримка реплікації забезпечує високу доступність та відмовостійкість.

3.2 Створення базової версії програмної системи аудиту пацієнтів

3.2.1 Розробка схеми бази даних

Розглянемо сутності, які будуть присутні в системі.

Ключовою сутністю в системі є пацієнт. Для пацієнта зберігатимемо його ім’я, дату народження, місто проживання, та історію його хвороб. Останнє поле зберігатиметься у вигляді строки в JSON форматі. Це поле відображає інформацію про історію хвороб пацієнта і є важливою для лікарів для вибору правильного лікування та діагностування. Воно може бути оновлене після відвідування лікаря, або через імпорт даних із зовнішньої системи.

Зовнішні системи – це системи, які схожі на ту, що створюється в рамках кваліфікаційної роботи та зберігають і обробляють дані про пацієнта. Може трапитись так, що в різних лікарнях або приватних клініках будуть встановлені різні системи і пацієнту доведеться кожного разу вводити свою інформацію при реєстрації. Це може займати багато часу, а також існує ризик забути внести якусь важливу інформацію про історію захворювань, що може негативно сказатися на точності діагнозу. Тому наша система матиме функціональність імпорту та експорту даних в сторонні системи. На ER діаграмі пацієнт (Patient) та стороння система (External System) зв’язані відношенням багато-до-багатьох через проміжну таблицю ExternalSystemMapping.

Іншою важливою сутністю є лікар (Doctor). У нашій системі ми зберігаємо лише ім’я лікаря та його спеціалізацію. В рамках дослідження інша інформація про лікаря для нас не важлива. Більше того, робиться спрощення та вважається, що лікар може мати лише одну спеціальність (наприклад кардіолог, чи офтальмолог), хоча на практиці лікар може мати декілька.

Спеціалізація (Specialization) винесена в окрему таблицю та зв’язана з лікарем зв’язком один-до-багатьох.

Ще одною важливою сутністю, яка фактично поєднує усі сутності системи воєдино є сутність Прийом (Appointment).Вона відображає запис на прийом пацієнта до лікаря. Ми зберігатимемо ідентифікатори лікаря та пацієнта, дату прийому, коментарій від пацієнта та діагноз – заключення лікаря після прийому.

В свою чергу Діагноз (Diagnosis) – це окрема сутність, яка містить в собі посилання на ще одну сутність Хвороба (Illness) та на сутність прийому. Так зроблено для того, щоб можна було продемонструвати аналіз статистики хвороб та прийомів.

В ході розробки моделі даних була створена ER діаграма, що зображена на рисунку 3.1 нижче.

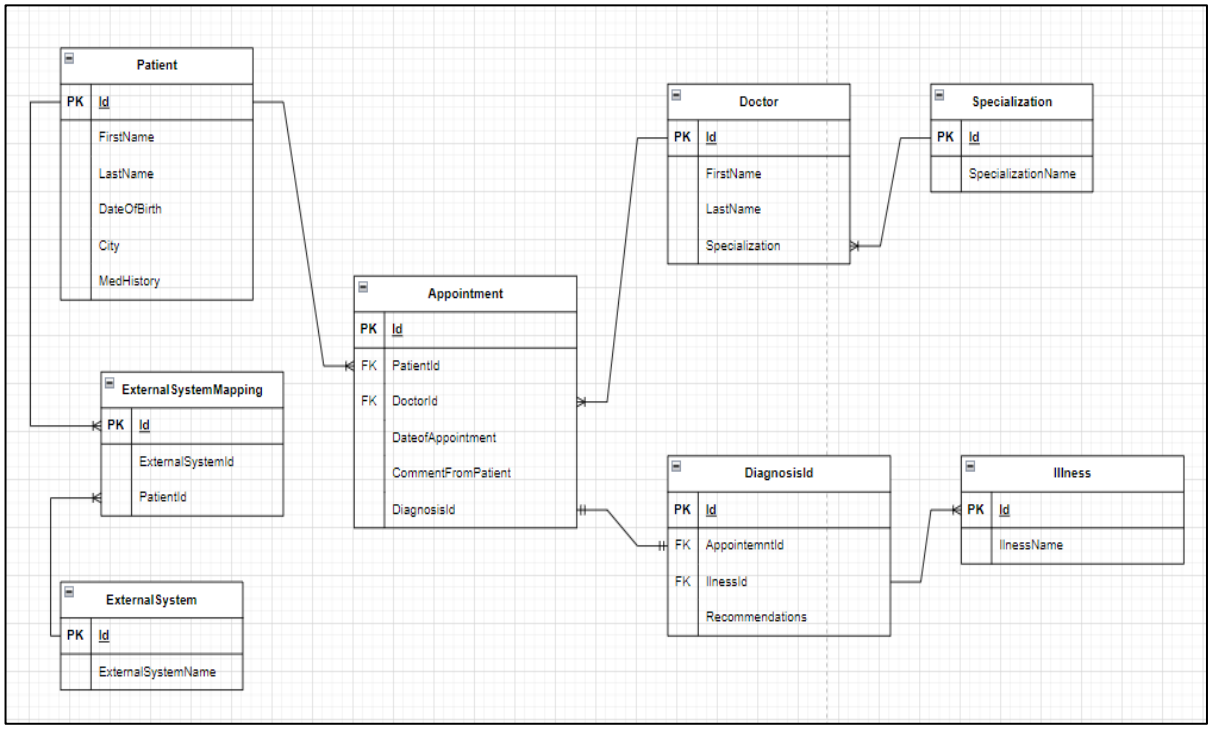

Рисунок 3.1 – ER діаграма (рисунок виконано самостійно)

Така модель даних, звичайно є спрощенням. В реальній системі все було б набагато складніше. Було би присутньо більше сутностей та атрибутів. Проте для цілей цього дослідження, такої системи цілком достатньо, щоб дослідити переваги і недоліки тих чи інших архітектурних підходів.

3.2.2 Розробка архітектури базової версії програмної системи

Як було сказано вище, за базову версію програмної системи візьмемо дуже просту монолітну трирівневу архітектуру. Бекенд-додаток складається з трьох рівнів: рівень даних, рівень бізнес-логіки, рівень презентації.

Рівень презентації відповідає за взаємодію з клієнтами нашого бекенду. Його задача у зручному вигляді (JSON форматі) представити для споживачів API необхідні дані. Також на цьому логічному рівні відбувається валідація усіх введених користувачем даних. Далі дані передаються на рівень бізнес-логіки для подальшої обробки. У випадку додатку, що буде розроблятися в рамках кваліфікаційної роботи, рівень презентації представлено контролерами, завдання який приймати HTTP запити від API клієнтів, моделями запитів та відповідей на запити та валідаторами, завдання яких перевіряти правильність вхідних даних.

Рівень бізнес-логіки, як випливає з назви, обробляє основну логіку програмної системи. Він є своєрідним мозком системи, де відбуваються основні обчислення та прийняття рішень. Тут відбувається обробка транзакцій та запитів від рівня презентації. Також цей рівень взаємодіє з рівнем даних для отримання необхідної інформації та збереження результатів.

Рівень даних призначений для взаємодії з сховищем даних. Там інкапсулювана робота з сутностями та ORM системами для забезпечення надійного та ефективного доступу до даних у базі даних.

Такий архітектурний підхід дозволяє розділити обов’язки між різними рівнями, полегшуючи при цьому розробку, тестування та підтримку програмного забезпечення.

На рисунку 3.2 представлена діаграма, виконана за допомогою сервісу draw.io, що демонструє базову монолітну архітектуру.

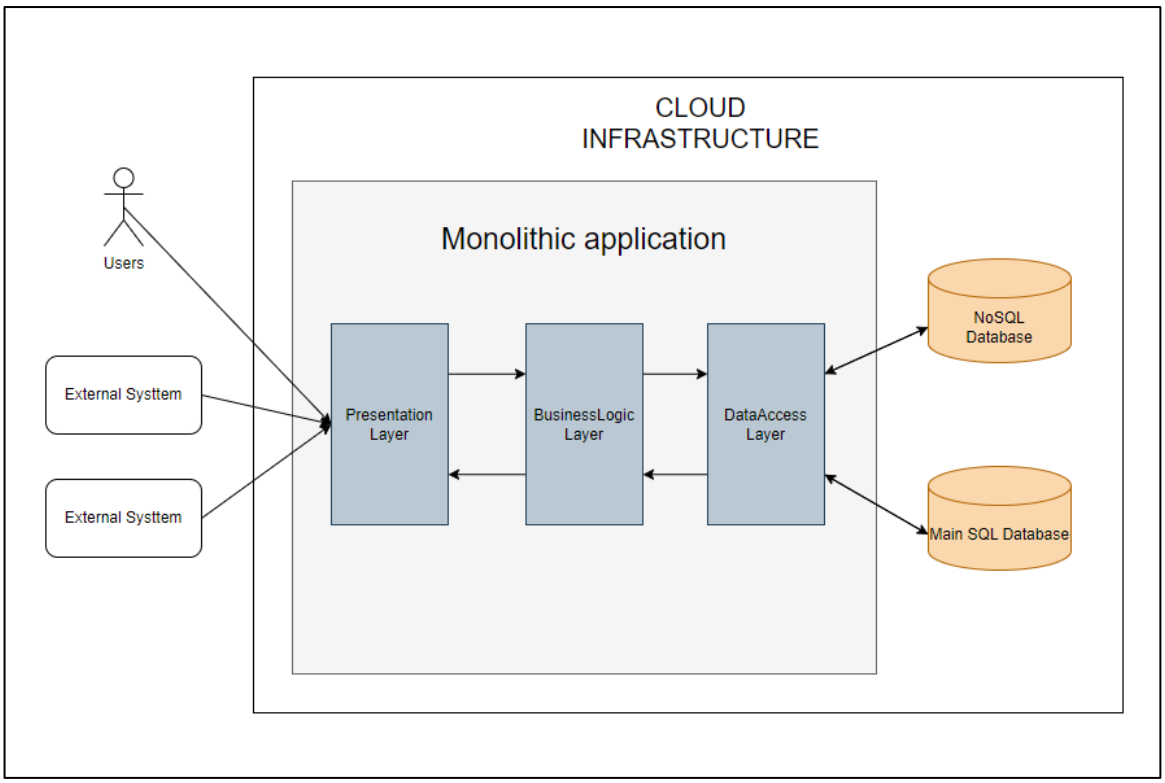

Рисунок 3.2 – Монолітна архітектура системи аудиту пацієнта (рисунок виконано самостійно)

3.2.3 Розгортання базової версії системи в AWS

Тепер розглянемо те, як дана версія системи розгорнута на AWS. Як вже згадувалося раніше, для більш зручного розгортання програмної системи буде використовуватись Docker. Для цього було створено докер файл, який в декларативному вигляді описує як саме докер має контейнеризувати додаток. Після контейнерезації зібраний образ буде відправлено в Docker Registry, звідки його потім можна буде скачати безпосередньо з серверів, де будуть розгортатись екземпляри системи.

Для розгортання контейнерів з додатком буде використовуватись AWS EC2 сервіс [21]. Це веб-сервіс, що надає масштабовані обчислювальні потужності в хмарі. Користувачі можуть запускати віртуальні сервери, відомі як інстанси,

вибираючи з різноманітних типів, що відповідають їхнім потребам у ресурсах, таких як процесор, оперативна пам'ять і сховище. EC2 дозволяє легко збільшувати або зменшувати потужності в залежності від навантаження додатків, забезпечуючи високу доступність та надійність. Інстанси EC2 інтегруються з іншими сервісами AWS, такими як S3 для зберігання, RDS для баз даних та CloudWatch для моніторингу.

Для більшої відмовостійкості будемо використовувати декілька однакових EC2 інстансів. Тобто декілька екземплярів бекенду програмної системи буде виконуватись одночасно. Якщо з один екземпляр перестане працювати, або не буде відповідати на запити – трафік буде перенаправлено на інші. Для балансування трафіку будемо використовувати Elastic Load Balancer (ELB). Його основне завдання розподіляти вхідний трафік між декількома EC2 інстансами, на яких підняті докер контейнери.

Для того, щоб підключити до нашої системи реляційну базу даних використаємо Amazon RDS. Amazon RDS (Relational Database Service) — це керований сервіс баз даних, який спрощує налаштування, управління та масштабування реляційних баз даних у хмарі. RDS підтримує кілька популярних систем управління базами даних, включаючи Amazon Aurora, PostgreSQL, MySQL, MariaDB, Oracle та SQL Server. Він автоматично виконує такі завдання, як резервне копіювання, оновлення програмного забезпечення та моніторинг продуктивності, що дозволяє користувачам зосередитися на розробці додатків. RDS забезпечує високу доступність і стійкість до збоїв за допомогою функцій автоматичного резервування та реплікації даних. Для нашої системи налаштуємо цей сервіс використовувати MS SQL Server.

Для розгортання нереляційної БД використаємо DynamoDB [22] сервіс. Це повністю керована, масштабована нереляційна база даних NoSQL, що забезпечує високу продуктивність і низьку затримку для додатків будь-якого масштабу. Вона підтримує моделі даних ключ-значення та документо-орієнтовані моделі, що дозволяє зберігати і обробляти великі обсяги неструктурованих даних. DynamoDB автоматично розподіляє дані і трафік між декількома серверами для забезпечення

високої доступності і надійності. Сервіс інтегрується з іншими AWS сервісами, такими як AWS Lambda для безсерверних обчислень і Amazon Kinesis для обробки потоків даних в реальному часі. Крім того, DynamoDB забезпечує можливості автоматичного масштабування і шифрування даних для підвищення безпеки.

Ще однією важливою вимогою для системи, що розробляється є можливості моніторингу та аудиту. Для моніторингу використаємо сервіс Cloud Watch. Детальніше про метрики моніторингу та аудиту буде розказано в наступних підрозділах.

Діаграма розгортання монолітної версії програмної системи приведена в Додатку А.

3.2.4 Розгортання базової версії системи в Microsoft Azure

Тепер розглянемо, як та сама версія системи була розгорнута в інфраструктурі іншого хмарного провайдера – Microsoft Azure.

Для запуску та виконання коду, що зібрано в docker контейнер, буде використовуватися сервіс Azure Virtual Machines. We сервіс хмарних обчислень, який надає можливість створення і використання віртуальних серверів у хмарі. Користувачі можуть запускати інстанси віртуальних машин, вибираючи з різноманітних конфігурацій апаратного забезпечення, які підходять для різних обчислювальних завдань. Віртуальні машини надають повний контроль над операційною системою, дозволяючи встановлювати і налаштовувати будь-яке програмне забезпечення [23]. Azure VMs підтримують різні операційні системи, включаючи Windows та Linux, і інтегруються з іншими сервісами Azure, такими як зберігання даних, мережі та моніторинг. Вони забезпечують масштабованість, високу доступність і можливість автоматизації через різноманітні інструменти управління, такі як Azure CLI, PowerShell та Azure Portal. В цілому, даний сервіс дуже схожий за своєю функціональністю на EC2 від AWS, і фактично є його аналогом.

Як і при розгортанні системи на AWS, буде створено декілька однакових екземплярів контейнеризованого додатку. Навантаження буде розподілятися між ними за допомогою Azure Load Balancer – басансувача навантаження, який пропонується даним хмарним провайдером. Більш детально зупинятися на цьому сервісі ми не будемо, адже за функціональністю він майже аналогічний Aplication Load Balancer від AWS.

Для розгортання баз даних використаємо сервіс Azure SQL Database і налаштуємо його використовувати MSSQL Server під капотом (для реляційної бази даних) та налаштуємо Azure Cosmos DB – нереляційну базу з підтримкою MongoDB для зберігання неструктурованих даних.

Як і у випадку з AWS, нам необхідний сервіс, що дозволить отримувати метрики, статистику та проводити аудит усіх сервісів, що використовуються при розгортанні системи. Для цього використаєм сервіс Azure Monitor – аналог Cloud Watch від AWS.

Діаграма розгортання системи в інфраструктурі Azure приведена в додатку Б.

3.3 Мікросервісна архітектура зі спільною базою даних

3.3.1 Проектування мікросервісної архітектури зі спільною базою даних

На попередньому етапі дослідження була створена базова версія архітектури програмної системи. Це була найпростіша монолітна архітектура без використання прийомів та засобів для захисту інформації. Наступним кроком буде покращення даної архітектури шляхом розділення її на мікросервіси за принципом функціональності та областей домену. Таким чином монолітний додаток буде розділений на декілька сервісів, кожен з який буде виконувати вузьконаправлені функції та відповідати за роботу лише з певними даними. При цьому, зазначимо те, що усі мікросервіси будуть використовувати одну спільну базу даних. Технічно, через це, ми не можемо назвати цю архітектуру повністю мікросервісною. Адже система все одно матиме єдину точку відмови – базу даних і сервіси так чи інакше будуть зв’язані одне з одним, хоча і не так тісно як в монолітному додатку. Сервіси

зможуть розроблятися та розгортатися окремо один від одного, хоча все одно зберігатиметься залежність на дані.

Перший сервіс, який було створено – це Authentication Service. Його задача забезпечувати перевірку прав користувачів на доступ до системи та видавати спеціальні JWT-токени, за допомогою яких клієнти отримуватимуть доступ до захищених ресурсів чи кінцевих точок.

Наступні два сервіси, які будуть розглянуті – це DoctorRegistration.Service і PatientRegistration.Service. Вони відповідають за реєстрацію нових користувачів в системі. Більшість ендпойнтів в цих сервісах потребують авторизації, тому запити до них так чи інакше мають проходити через Authentication.Service для підтвердження прав користувача на виконання тих чи інших дій.

Appointment Service – відповідає за управління візитами пацієнтів до лікарів. За допомогою цього сервісу створюються сутності апойнтментів, лікарі та пацієнти отримуватимуть актуальну інформацію про стан відвідування, можуть відміняти та переносити візит тощо. За допомогою цього сервісу лікарі вносять інформацію про діагноз та рекомендації пацієнту після візиту.

Statistics.Service відповідає за отримання статистичних даних щодо пацієнтів, хвороб, лікарів, діагнозів тощо.

Appointment.Service і Statistics.Service не взаємодіють з базою даних напряму. Він робить це за допомогою внутрішнього PatientData.Service. Введемо поняття внутрішній і зовнішній сервіс. Зовнішнім (або публічним) називається сервіс з яким напряму можуть взаємодіяти клієнти бекенд додатку. Взаємодія між такими сервісами обов’язково потребує аутентифікації. Адже їх IP адреси публічні і будь-хто може посилати запити. З іншого боку, внутрішні сервіси – це сервіси які не мають публично доступної IP адреси, або екрановані за допомогою VPN. Запити до них можуть слати лише інші сервіси з цієї системи (або довірені IP адреси). Тому в деяких архітектурах такі сервіси не потребуватимуть аутентифікації, адже сам факт того, що викликаюча сторона змогла здійснити виклик говорить про те, що вона мала на це право.

Усі описані вище сервіси (окрім PatientData.Service) є публічними тому потребуватимуть звернення до Authentication.Service задля видачі токену з правами доступу. Усі ці сервіси так чи інакше взаємодіють зі спільною базою даних.

Іншим піддоменом системи є Integration.Service. Його основна задача синхронізувати дані пацієнта в нашій системі з даними інших системах обліку. Це відбувається за допомогою приватних Import і Export сервісів, які забезпечують функціональність імпорту та експорту даних відповідно. В свою чергу ці сервіси взаємодіють із вже відомим нам PatientData.Service для отримання доступу до даних пацієнтів. Аутентифікація/Авторизація вимагається тільки для Integration.Service.

Ще слід сказати, що Import.Service записує проміжні дані імпорту в нереляційну базу даних для забезпечення того, що імпорт буде продовжено, навіть якщо по якійсь причині його буде перервано.

На рисунку 3.3 приведена діаграма взаємодії мікросервісів між собою та зі сховищем даних.

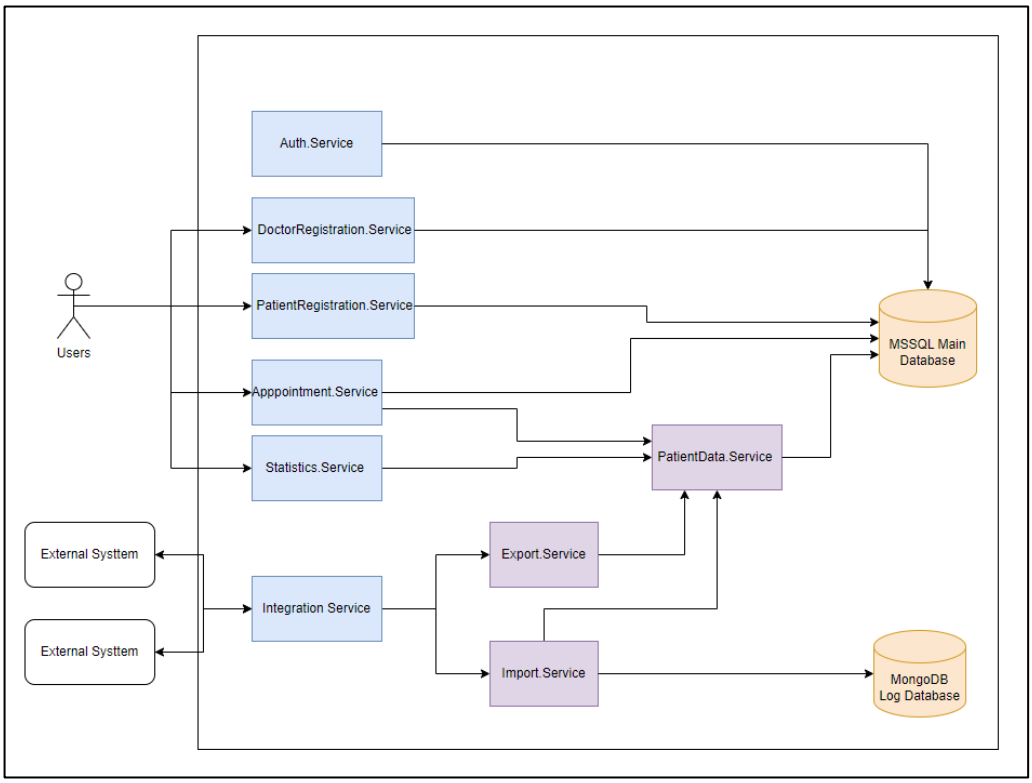

Рисунок 3.3 – Архітектура мікросервісною версії програмної системи зі спільною базою даних (рисунок виконано самостійно)

3.3.2 Розгортання системи з мікросервісною архітектурою за спільними базами даних на AWS

Розглянемо розгортання цієї версії архітектури в інфраструктурі хмарного провайдера AWS. Як вже згадувалося в розділах вище, сервіси можна поділити на приватні та публічні за їх доступністю в глобальному інтернеті. Для розгортання і конфігурації сервісів з потрібною доступністю скористаємось Amazon Virtual Private Cloud (VPC). Amazon Virtual Private Cloud дозволяє створювати ізольоване мережеве середовище в AWS, де ви можете запускати свої ресурси, такі як EC2 інстанси. Використовуючи VPC, ви можете створювати як публічні, так і приватні підмережі. Публічні підмережі мають маршрутизацію до Інтернету через Internet Gateway і використовуються для розміщення публічних сервісів, таких як веб-сервери або Load Balancers. Приватні підмережі не мають прямого доступу до Інтернету і використовуються для розміщення внутрішніх сервісів, таких як бази даних або бекенд-сервіси. Для забезпечення вихідного доступу з приватних підмереж до Інтернету (наприклад, для оновлень), ви можете використовувати NAT Gateway, розміщений у публічній підмережі. Це забезпечує безпечне та контрольоване мережеве середовище для ваших додатків, дозволяючи розділяти публічні та приватні ресурси в межах одного VPC [24].

Таким чином, сервіси які мають бути публічними разом з їхніми балансувачами навантаження будуть розміщені в приватній підмережі та матимуть доступ в інтернет. Усі приватні сервіси, а також бази даних будуть розміщені в приватні підмережі, та будуть доступні тільки зсередини VPC. Таким чином, нам вдасться забезпечити безпеку внутрішніх червісів. Вони не потребуватимуть додаткової аутентифікації, так як доступ до них матимуть лише сервіси з мережі.

Повна діаграма розгортання системи з даною архітектурою приведена в додатку В.

Аналогічний розділ для Microsoft Azure не приводиться, адже розгортання майже ідентичне. Відрізняються тільки назви сервісів та ресурсів.

3.4 Мікросервісна архітектура з окремими базами даних

3.4.1 Проектування мікросервісної архітектури з окремими базами даних

Попередня версія архітектури програмної системи має один суттєвий недолік – єдину точку відмови. Так як усі мікросервіси використовують спільну базу даних, то у випадку недоступності цієї бази даних уся система буде не доступна. Окрім цього, єдина база даних створює ризик, що у разі, якщо зловмисник отримає доступ до неї – він отримає доступ одразу до усіх даних системи. Єдина база даних також є не дуже гнучким рішенням через те, що ми не можемо налаштовувати різні політики зберігання для різних типів даних.

З цими проблемами буду боротися наступне архітектурне рішення. Кожен мікросервіс матиме окрему базу даних і буде взаємодіяти з нею та лише з нею. Таким чином, у випадку недоступності одного сервісу, на функціональність інших не вплине. Це в свою чергу підвищить доступність та гнучкість системи.

Кожна база даних буде зберігати лише ті дані, що необхідні для роботи певного мікросервісу. Буде присутнє певне дублювання даних між базами, адже декільком мікросервісам можуть бути потрібні одні й ті самі дані (наприклад дані про пацієнта). Таким чином декілька БД можуть зберігати ці дані. Вникає наступне питання: «Як забезпечити синхронізацію цих даних?». Тобто, якщо дані оновлюються в одному місці як забезпечити оновлення даних в усіх інших БД. Рішень може бути декілька. Два основних підходи до синхронізації даних – це синхронний та асинхронний.

При синхронному способі синхронізації мікросервіс за допомогою HTTP протоколу викликає всі інші сервіси з командою оновити дані. Він чекає поки інші сервіси зроблять оновлення, повернуть відповідь і лише потім вважає запит повністю виконаним. З одного боку такий підхід забезпечує гарантію того, що дані будуть оновлені в усіх місцях, або не будуть оновлені ніде, якщо щось піде не так – забезпечується атомарність операції між мікросервісами. При використанні синхронного підходу дані завжди знаходяться в актуальному стані, проте операції

оновлення займають багато часу. І якщо один з сервісів недоступний уся операція оновлення не може бути виконана [25].

При асинхронному підході використовується брокер-повідомлень та publisher-subscriber патерн. Сервіс який ініціює оновлення публікує повідомлення про це на брокер, а інші сервіси, якім також треба оновити ці дані підписані на цю подію і коли вона виникає виконують оновлення в себе. Ця вся взаємодія відбувається асинхронному режимі, а мессадж-брокер забезпечує довготривале зберігання повідомлення допоки усі не отримають на не оброблять його. Таким чином він забезпечує гарантовану доставку. Таким чином навіть якщо сервіс-підписник в певний час не доступний, він зможе обробити це повідомлення пізніше, коли знов буде працювати. До недоліків такого підходу можна віднести той факт, що стан даних в окремо взятому мікросервісі може бути не найактуальніший через природу асинхронної взаємодії. Для даної системи використаємо саме асинхронний підхід.

У якості брокера повідомлень для AWS використаємо SQS (Simple Queue Service). Amazon Simple Queue Service (SQS) – це керована черга повідомлень, яка дозволяє ізолювати компоненти хмарних додатків для забезпечення масштабованості та високої доступності. SQS підтримує стандартні черги для високої пропускної здатності та черги FIFO для забезпечення порядку обробки повідомлень. Сервіс дозволяє передавати великі обсяги даних між компонентами додатків, інтегруючись з іншими сервісами AWS, такими як Lambda, EC2 та S3. SQS забезпечує безпеку завдяки шифруванню даних та керуванню доступом через IAM політики.

В свою чергу у Microsoft Azure є Azure Queue Storage [26]. Це сервіс для надійного зберігання та передачі великих обсягів повідомлень між компонентами додатків. Azure Queue Storage дозволяє створювати масштабовані розподілені системи з асинхронною обробкою завдань, забезпечуючи надійність і високу доступність повідомлень. Як і SQS, Azure Queue Storage інтегрується з іншими сервісами Azure, такими як Azure Functions, Virtual Machines, і Blob Storage, що дозволяє створювати гнучкі та надійні архітектури додатків.

Ще один архітектурний прийом, що буде використано для забезпечення більшої безпеки та гнучкості – це API Gateway. Використання патерну API Gateway у мікросервісній архітектурі надає кілька важливих переваг. API Gateway виступає єдиною точкою входу для клієнтів, спрощуючи керування та захист доступу до мікросервісів. Цей патерн дозволяє централізовано реалізовувати такі функції, як аутентифікація, авторизація, маршрутизація запитів, агрегація відповідей та обмеження швидкості запитів, знижуючи складність мікросервісів. API Gateway допомагає оптимізувати продуктивність системи, зменшуючи кількість запитів між клієнтом і серверами та забезпечуючи кешування часто запитуваних даних.

Буде створений ще один сервіс Api Gateway Service, що буде слугувати єдиною точкою входу в систему. Він виступатиме єдиним публічним сервісом в системі. Усі інші сервіси будуть приватними і доступ до них буде лише з IP адреси API Gateway сервісу. Повну діаграму аріхтектури такої архітектури в інфраструктурі AWS приведено на рисунку 3.4

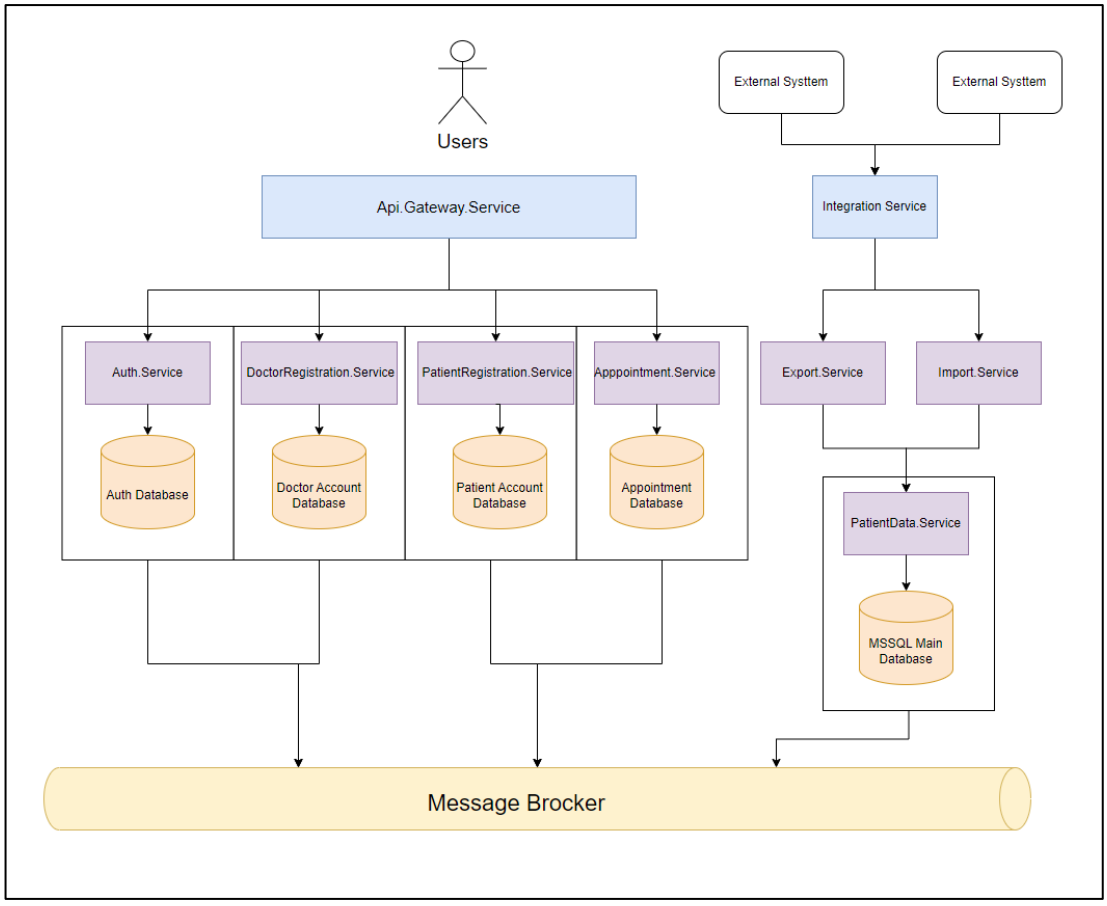

Рисунок 3.4 – Діаграма мікросервісної архітектури з окремими базами даних (рисунок виконано самостійно)

3.4.2 Розгортання мікросервісної архітектури з окремими базами даних

Розглянемо розгортання цієї версії архітектури в інфраструктурі хмарного провайдера AWS. Як і в попередній версії архітектури програмної системи буде використано Amazon VPC та публічні та приватні підмережі. Основною відмінністю від попередньої версії буде те, що тепер кожен сервіс матиме окрему базу даних, куди матиме доступ лише він і тільки він. Як вже згадувалось раніше, для зв’язку сервісів між собою і синхронізації даних буде використовуватись асинхронний підхід з брокером повідомлень Таким брокером виступатиме Simple Queue Service.

Повна діаграма розгортання цієї архітектури приведена в додатку Г.

3.5 Використання підходу Zero Trust

Наступною трансформацією архітектури системи буде впровадження принципів Zero Trust в архітектуру. Модель Zero Trust є сучасним підходом до кібербезпеки, який базується на принципі "ніколи не довіряй, завжди перевіряй". На відміну від традиційних методів, які покладаються на периметрову безпеку, Zero Trust вимагає автентифікації та авторизації кожного користувача та пристрою незалежно від їх розташування. Це означає, що всі користувачі, пристрої та додатки постійно перевіряються, і доступ до ресурсів надається тільки на основі чітко визначених політик.

Основні принципи Zero Trust включають багатофакторну аутентифікацію (MFA), мінімальні привілеї (Least Privilege) та постійний моніторинг. Багатофакторна аутентифікація вимагає кількох етапів підтвердження особи користувача, що значно знижує ризик несанкціонованого доступу. Принцип мінімальних привілеїв гарантує, що користувачі мають доступ лише до тих ресурсів, які необхідні для виконання їхніх завдань, що зменшує можливість зловживання [27].

Сегментація мережі та мікросегментація забезпечують додатковий рівень безпеки, обмежуючи поширення загроз у разі компрометації одного з сегментів. Постійний моніторинг і аналітика дозволяють виявляти підозрілу активність та швидко реагувати на потенційні загрози.

Використання Zero Trust у хмарному середовищі передбачає інтеграцію з такими сервісами, як AWS IAM або Azure Active Directory для управління ідентифікацією та доступом, а також AWS CloudWatch або Azure Monitor для моніторингу. Шифрування даних за допомогою сервісів на кшталт AWS KMS або Azure Key Vault забезпечує додатковий захист даних.

Модель Zero Trust дозволяє значно підвищити рівень безпеки організацій, зменшити ризики компрометації даних та забезпечити гнучке управління доступом. Впровадження цього підходу є важливим кроком у забезпеченні захисту в сучасному цифровому середовищі.

На попередньому етапі ми вже розділили сервіси на приватні та публічні, використавши VPC та підмережі. Тепер на кожен запит до захищеного ресурсу буде відбуватись, по-перше, перевірка прав даного користувача/аккаунту отримати доступ до захищеного ресурсу, по-друге, перевірка прав сервісу отримати доступ до захищеного ресурсу.

У першому випадку використовуватиметься ASP.NET Core Auth.Service. Який перевірятиме доступ та видаватиме новий JWT токен з дуже коротким терміном життя. Близько 30 секунд. Цього терміну життя в абсолютній більшості випадків буде достатньо, щоб виконати один запит. Після закінчення цього терміну, токен має бути отримано заново. З одного боку, такий підхід вносить додаткову складність і збільшує час виконання запиту, проте має забезпечити найкращу безпеку.

3.6 Використання шифрування даних

Наступним і останнім кроком для трансформації нашої архітектури – буде впровадження шифрування даних.. Шифрування даних є критично важливим

компонентом для забезпечення безпеки в сучасних програмних системах, особливо у хмарному середовищі. У нашій системі, яка працює з конфіденційними даними пацієнтів, застосовуються два основних види шифрування: шифрування даних в спокої (at rest) та шифрування при передачі (in transit).

Шифрування даних в спокої забезпечує захист інформації, яка зберігається в базах даних, файлових системах або інших сховищах. Для цього ми використовуємо потужні алгоритми шифрування, такі як AES-256. У AWS це може бути реалізовано за допомогою сервісів AWS Key Management Service (KMS), який надає централізоване управління ключами шифрування. Дані в Amazon RDS та DynamoDB також можуть бути автоматично зашифровані з використанням KMS. В Azure аналогічні можливості надає Azure Key Vault. Кожен запис, що зберігається у базах даних або сховищах, буде зашифрований, що унеможливлює доступ до них у разі несанкціонованого доступу до фізичних носіїв або резервних копій.

Шифрування при передачі [28] гарантує, що дані залишаються захищеними під час їх переміщення між клієнтами, серверами та різними компонентами системи. Для цього використовуються протоколи шифрування, такі як TLS (Transport Layer Security). Всі запити до нашого API, що працює на ASP.NET Core, будуть зашифровані за допомогою TLS, що запобігає можливим атакам типу "man-in-the-middle". AWS забезпечує автоматичне шифрування даних під час передачі між сервісами, такими як Amazon S3, Amazon RDS та іншими, через використання HTTPS. Аналогічно, в Azure шифрування під час передачі даних забезпечується через використання TLS/SSL.

Для управління ключами шифрування в обох хмарних середовищах використовуються централізовані служби управління ключами, що дозволяє спростити процес ротації ключів, управління доступом та аудиту. Це допомагає підтримувати високий рівень безпеки і забезпечувати відповідність регуляторним вимогам, таким як GDPR або HIPAA.

Управління ключами включає створення, зберігання та контроль доступу до криптографічних ключів, що є невід'ємною частиною нашої стратегії безпеки.

Регулярна ротація ключів та моніторинг доступу до них дозволяє мінімізувати ризики, пов'язані з компрометацією ключів.

Загалом, реалізація шифрування даних в спокої та при передачі в нашій системі забезпечує високий рівень захисту конфіденційної інформації пацієнтів. Це критично важливо для підтримки довіри користувачів і дотримання нормативних вимог. Завдяки інтеграції з хмарними сервісами, такими як AWS та Azure, ми можемо ефективно управляти шифруванням і підтримувати безпеку нашої системи на найвищому рівні.

3.7 Опис метрик для порівняння різних версій архітектур програмної системи

Для чисельного порівняння ефективності того чи іншого архітектурного рішення, необхідно скласти перелік метрик, значення яких будуть порівнюватись та з’ясувати як саме мають бути отримані ці значення.

Спочатку розглянемо метрики, що будуть використовуватись для оцінки продуктивності системи. Для цього нам потрібно отримувати статистику з усіх хмарних сервісів, при експлуатації системи. Нижче буде приведено повний список метрик, що будуть заміряні з описом того, що саме ця метрика означає, з якого сервісу та яким чином вона отримана.

Virtual Machines.

Для отримання віртуальних машин використовуються сервіси AWS EC2 та Azure Virtual Machines. Для отримання метрик з віртуальних машин, на яких будуть виконуватись докер-контейнери використаємо Cloud Watch та Azure Monitor відповідно. Будемо отримувати наступні метрики:

- CPU Utilization – відсоток використання процесора;
- Memory Utilization – відсоток використання пам’яті.

Так як одночасно буде запущено декілька інстансів додатку, значення цих метрик буде братися як середнє арифметичне від значень на усіх віртуальних машинах.

Load Balancer.

Для балансування навантаження між різними інстансами програмної системи використовується AWS ALB та Azure LB відповідно. Значення метрик отримаємо за допомогою Cloud Watch та Azure Monitor. Метрики наступні:

- Request Count – кількість запитів, що обробляються балансувальником навантажень;
- Latency – затримка в роботі запитів.

ASP NET Core додаток.

Для отримання метрик в ASP.NET Core можна використовувати вбудовані засоби логування, middleware для діагностики та механізми перевірки стану (health checks). Ці інструменти дозволяють відстежувати HTTP-запити, час відповіді, статусні коди та інші ключові показники продуктивності. Для інтеграції з Prometheus використовують бібліотеки, такі як prometheus-net, які дозволяють експортувати метрики у форматі, сумісному з Prometheus. Встановлення цієї бібліотеки та її налаштування у Startup.cs дозволяє збирати кастомні метрики та експортувати їх через HTTP-ендпоінт. Prometheus потім може зчитувати ці метрики та візуалізувати їх у графіках і дашбордах для подальшого аналізу.

Будемо отримувати наступні метрики:

- Request Count – кількість запитів, що обробляються додатком;
- Error Count – кількість помилок у додатку;
- Response Time – час відповіді на запити.

Окрім специфічних метрик для кожного окремого хмарного сервісу, що використовується системою, для оцінки ефективності системи важливо отримати загальну інформацію під час її роботи. Таким чином будуть фіксуватися:

- середній час відповіді додатку – вимірювання затримки та час обробки запитів;
- пропускна здатність – кількість запитів, що оброблюються за одиницю часу;
- час безвідмовної роботи;
- середній час відновлення після збоїв.

Усі перелічені метрики будуть аналізуватися під час звичайної роботи системи та під час роботи системи під навантаженням. Для такого тестування буде використовуватись інструмент Apache JMeter. Apache JMeter – це інструмент для навантагового тестування та вимірювання продуктивності веб-додатків. Він дозволяє створювати, виконувати та аналізувати тести для оцінки поведінки системи під різними навантаженнями. JMeter підтримує тестування різних типів серверів, включаючи веб-сервери, бази даних, FTP-сервери, веб-сервіси та інші. Завдяки своєму інтуїтивно зрозумілому інтерфейсу та широкому набору функціональних можливостей. JMeter можна використовувати для вимірювання продуктивності та стійкості різних реалізацій системи. Це дозволить об'єктивно оцінити, як різні архітектури впливають на продуктивність під навантаженням, а також виявити можливі точки відмови або вузькі місця в системі.

3.8 Опис безпекових тестів

Для забезпечення надійного захисту інформаційних систем і запобігання можливим кіберзагрозам критично важливим є створення безпекових тестів. Мета безпекових тестів полягає в ідентифікації, аналізі та усуненні можливих загроз для забезпечення захисту даних та ресурсів.

На етапі планування потрібно визначити цілі та обрати методи тестування. Будемо враховувати специфіку кожного сервісу та їх взаємодію.

Перший метод безпекового тестування, який буде використано – це статистичний аналіз коду.

Статичний аналіз коду (Static Code Analysis) — це процес перевірки вихідного коду програми без його виконання. Основна мета — виявити потенційні вразливості, помилки або недоліки в коді. На сьогоднішній день існує багато інструментів аналізу коду. Одним з найпопулярніших, що підтримує широкий спектр функціональностей для розробки на .NET, є ReSharper від JetBrains. Так як для розробки програмної системи використовувалася IDE Rider, то

функціональність ReSharpera вбудована, тому цей варіант інструменту статичного аналізу коду ідеально підходить для даного дослідження.

Треба зазначити, що статичний аналіз коду є важливим кроком процесу CI/CD і в більшості випадків має виконуватися автоматично. Наразі для аналізу створеного проекта встановимо ReSharper у якості розширення середи розробки. Це дасть можливість швидкого проведення аналізу усього рішення (меню ReSharper > Inspect > Code Issues in Solution).

Після проведеного аналізу потрібно звернути увагу на знайдені вразливості, такі як SQL-ін'єкції, XSS, небезпечні API, некоректна обробка даних, можливі витоки пам'яті та інші.

Друга техніка безпекового тестування, яка буде використана – це фаззинг.

Фаззинг (Fuzz Testing) — це техніка тестування, при якій в програму вводяться випадкові, некоректні або несподівані дані з метою виявлення помилок і вразливостей.

Визначимо, які частини системи будуть піддані фаззингу:

- форма реєстрації докторі;
- форма реєстрації пацієнтів;
- запити до сервісу прийому до лікаря;
- API для інтеграції з зовнішніми системами;
- механізми автентифікації та авторизації.

Згенеруємо випадкові та некоректні вхідні дані. Будемо використовувати спеціальні інструменти, такі як OWASP ZAP або Burp Suite.

Запустимо фазз-тестування для кожного з цільових компонентів:

- введення випадкових даних у поля форм реєстрації та логіну;
- відправка випадкових та некоректних даних до сервісу прийому до лікаря (дата, час, ідентифікатор лікаря, ідентифікатор пацієнта);
- відправка некоректних запитів до API, включаючи випадкові значення параметрів, недійсні токени автентифікації тощо.

Третім видом безпекового тестування, що буде проводитись є тестування на проникнення.

Тестування на проникнення (Penetration Testing або Pentesting) — це метод оцінки безпеки комп'ютерних систем, мереж або веб-додатків шляхом моделювання атаки зловмисника.

Нижче визначимо, які компоненти системи будуть тестуватися за допомогою яких методів та інструментів.

Будуть виконані наступні дії:

- проведемо тестування сервісів реєстрації докторів та пацієнтів, використавши SQL-інєкції та введення XSS скриптів у поля вводу форм реєстрації;
- спробуємо виконання дій від імені зареєстрованих користувачів через CSRF-атаки;
- виконаємо API-інєкції через запити до сервісу інтеграції;
- проведемо MitM атаки: перехоплення трафіку між системами для перевірки можливості модифікації даних;
- проведемо Brute-force атаки з використанням інструментів для підбору паролів;
- для виявлення відкритих портів і служб проскануємо мережу за допомогою Nmap.

Описані вище безпекові тести допоможуть забезпечити комплексний підхід до аналізу безпеки та якості коду створюваної системи, виявивши та усунувши потенційні вразливості як на етапі розробки, так і під час виконання системи.

4 ПРОВЕДЕННЯ ЕКСПЕРЕМЕТАЛЬНОГО ДОСЛІДЖЕННЯ

4.1Тестування на швидкість виконання запитів

Тепер, коли в нас спроектовано 6 версій архітектур однієї і тієї ж програмної системи, створено прототипи системи на кожну з п’яти архітектур та розміщено всі ці версії на інфраструктурі двох різних хмарних провайдерів ми можемо приступити до тестування і порівняння. Отже фактично в нас є 12 версій для порівняння.

Перший тест який ми проведемо – це відтворення сценарію реєстрації пацієнта та його запис на прийом до лікаря. Для підготовки до тестування нам необхідно створити сутність лікаря, щоб можна було виконати запис до нього. Зробимо це заздалегідь за допомогою інструменту Postman. Тепер, коли, лікаря створено ми можемо приступати до тестування. За сценарієм ми маємо створити сутність пацієнта і одразу після цього записатися на прийом до лікаря. Ендпойнти для виконання цих дій захищені авторизацією, тому спочатку треба отримати JWT токен з Auth.Service. Фактично, для першого тесту нам треба виконати три послідовних запити:

- api/patient/register;
- api/auth/login;
- api/appointment/createAppointment.

За допомогою фреймворку для автоматизованого тестування Selenium виконаємо ці запити до кожної з 12 версій програмних систем та зафіксуємо швидкість виконання сценарію і порівняємо її для різних версій. Назвемо цей тест «Тест 1.1».

Розглянемо другий тест кейс – сценарій перегляду списку лікарів та запису на повторний прийом (Тест 1.2)

Для підготовки до тестування нам необхідно створити сутність лікаря, щоб можна було його переглянути у списку. Зробимо це заздалегідь за допомогою інструменту Postman. Тепер, коли лікаря створено, ми можемо приступати до

тестування. За сценарієм ми маємо отримати список лікарів та записатися на повторний прийом до одного з них. Ендпойнти для виконання цих дій захищені авторизацією, тому спочатку треба отримати JWT токен з Auth.Service. Фактично, для другого тесту нам треба виконати чотири послідовних запити:

- api/auth/login;
- api/doctor/list;
- api/appointment/createAppointment (для першого прийому);
- api/appointment/createAppointment (для повторного прийому).

Третій сценарій - сценарій редагування профілю пацієнта та скасування запису на прийом (Тест 1.3).

Для підготовки до тестування нам необхідно створити сутність пацієнта та лікаря, щоб можна було виконати запис до лікаря, а потім скасувати цей запис. Зробимо це заздалегідь за допомогою інструменту Postman. Тепер, коли лікаря та пацієнта створено, ми можемо приступати до тестування. За сценарієм ми маємо виконати авторизацію, відредагувати профіль пацієнта, записатися на прийом до лікаря, а потім скасувати цей запис. Фактично, для третього тесту нам треба виконати шість послідовних запитів:

- api/auth/login;
- api/patient/updateProfile;
- api/appointment/createAppointment;
- api/appointment/list (перегляд списку записів);
- api/appointment/cancelAppointment (скасування запису);
- api/appointment/list (перегляд списку записів після скасування).

Для проведення кожного з цих тестів використовуватимемо інструмент Postman, де спочатку отримаємо JWT токен з Auth.Service, а потім виконаємо відповідні послідовні запити до захищених ендпойнтів. Це дозволить нам перевірити, що всі процеси авторизації та бізнес-логіки працюють належним чином.

Після виконання усіх тест кейсів на усіх версіях програмної системи була отримана таблиця результатів з часами виконання усіх сценаріїв. Вона приведена

на рисунку 4.1. За даними з цієї таблиці були побудовані графіки, що приведені на рисунках 4.2 та 4.3 та 4.4 відповідно.

Provider	Architecture	Test 1.1	Test 1.2	Test 1.3
AWS	Monolith	213 ms	324 ms	1056 ms
AWS	Microservices Shared DB	353 ms	397 ms	1323 ms
AWS	Microservices Separate DB	305 ms	406 ms	1345 ms
AWS	Zero Trust	454 ms	563 ms	1700 ms
AWS	Encryption	341 ms	417 ms	1349 ms
AWS	Zero Trust + Encryption	561 ms	650 ms	1984 ms
Azure	Monolith	200 ms	342 ms	1056 ms
Azure	Microservices Shared DB	350 ms	351 ms	1221 ms
Azure	Microservices Separate DB	353 ms	435 ms	1342 ms
Azure	Zero Trust	574 ms	665 ms	1854 ms
Azure	Encryption	361 ms	453 ms	1421 ms
Azure	Zero Trust + Encryption	604 ms	708 ms	1884 ms

Рисунок 4.1 – Результати тестування на швидкість виконання запитів (рисунок виконано самостійно)

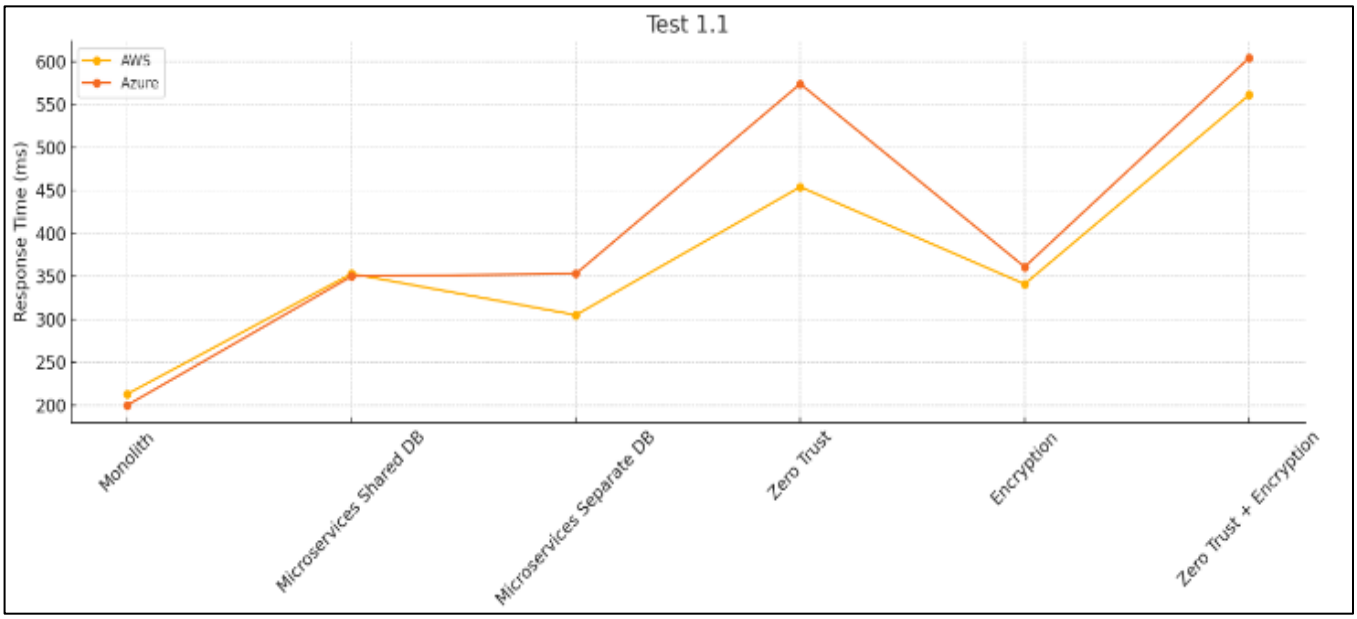

Рисунок 4.2 – Результати тесту 1.1 (рисунок виконано самостійно)

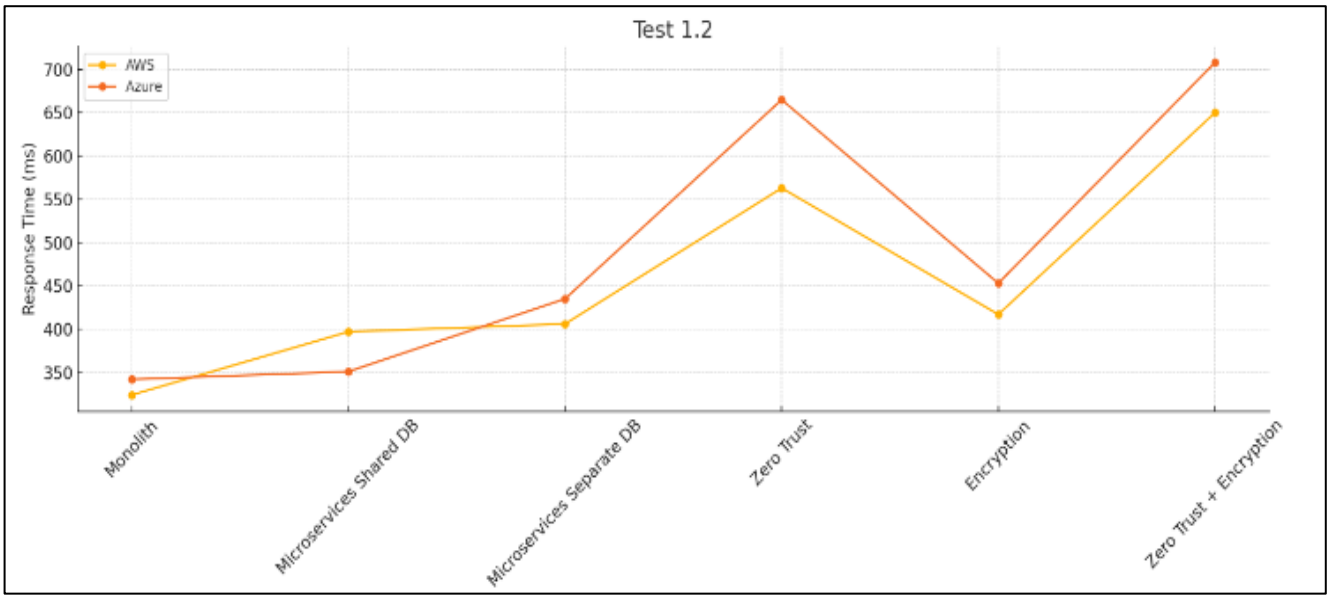

Рисунок 4.3 – Результати тесту 1.2 (рисунок виконано самостійно)

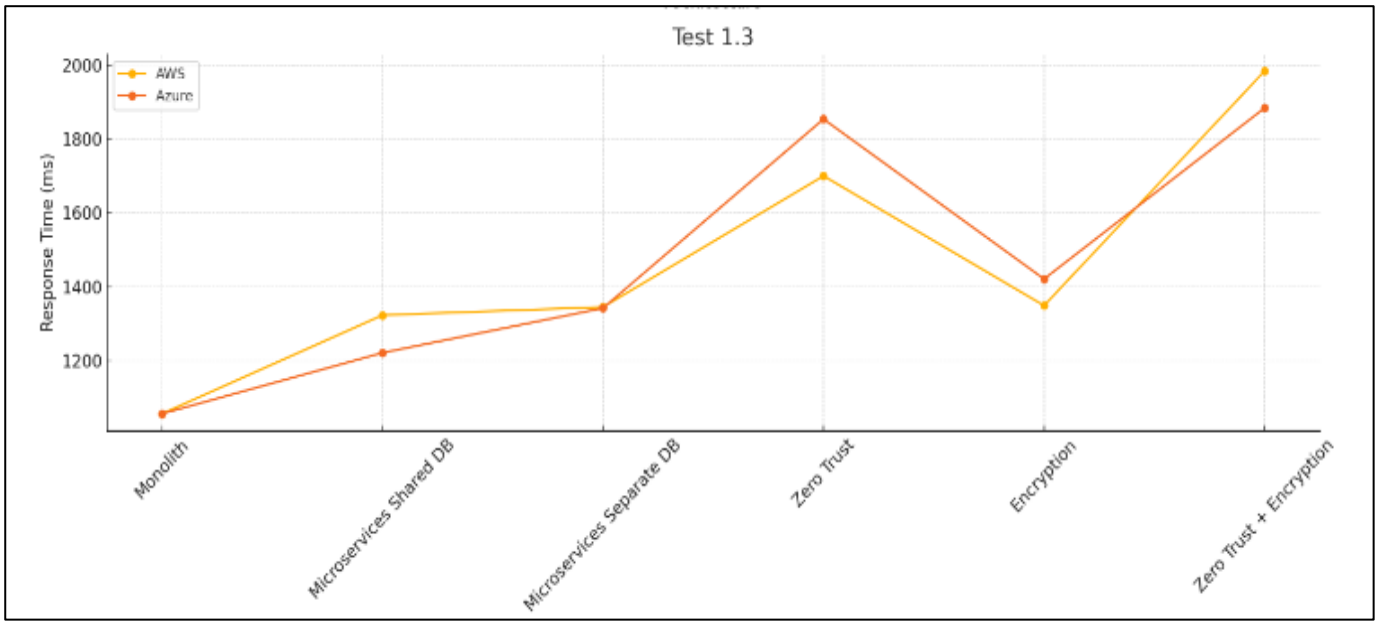

Рисунок 4.4 – Результати тесту 1.3 (рисунок виконано самостійно)

Бачимо, що графіки для Microsoft Azure та AWS майже ідентичні, тому можемо зробити попередній висновок, що вибір хмарного провайдера (з провідних провайдерів) не впливатиме на час відповіді на одиничний запит. В цей же час бачимо, що використання Zero Trust моделі сильно збільшує час відповіді на запит, тому сильно впливає на продуктивність.

4.2 Тестування навантаженням

Наступний вид тестування, який буде проведено – це тестування навантаженням. Для цього використаємо інструмент Apache JMeter, про який вже згадувалося раніше.

Для тестування оберемо запит, виконання якого займає найбільше ресурсів та часу. Це буде запит на отримання повної статистики по використанню системи користувачами. Ендпойнт повертає інформацію про пацієнтів, лікарів, аппойнтменти та деталі діагнозів. Ендпойнт спеціально спроектовано таким чином, щоб для виконання цього запиту необхідно було запросити інформацію одразу з декількох сервісів. Через це, виконання цього запиту є досить ресурсозатратним.

Спочатку, виконаємо не велику кількість запитів (10 запитів одночасно) до кожної з версій програмної системи, щоб мати змогу вирначити еталонний час виконання запиту та інші показники. Результати приведені на рисунку 4.5.

Provider	Architecture	Base Test (10 requests)				
		Average Response Time (ms)	CPU Average (%)	Memory Average (Mb)	Latency (ms)	Error Count
AWS	Monolith	413	15	500	105	0
AWS	Microservices Shared DB	553	25	750	150	0
AWS	Microservices Separate DB	505	20	700	140	0
AWS	Zero Trust	654	30	850	200	0
AWS	Encryption	501	33	720	160	0
AWS	Zero Trust + Encryption	761	35	900	250	0
						0
Azure	Monolith	400	12	480	100	0
Azure	Microservices Shared DB	550	23	730	145	0
Azure	Microservices Separate DB	563	20	690	135	0
Azure	Zero Trust	774	32	780	210	0
Azure	Encryption	561	36	750	170	0
Azure	Zero Trust + Encryption	804	41	830	260	0

Рисунок 4.5 – Результати еталонного тестування навантаженням (рисунок виконано самостійно)

Тепер приступимо саме до тестування під навантаженням. За допомогою Apache JMeter виконаємо 100 (Тест 2.1), 1000 (Тест 2.2) та 10000 (Тест 2.3) запитів до кожної з різних версій програмних систем. На рисунку 4.6 зображено скріншот

інтерфейсу Jmeter при проведення тестування навантаженням. На рисунку приведена статистика по запитам, що були виконані під час тестування.

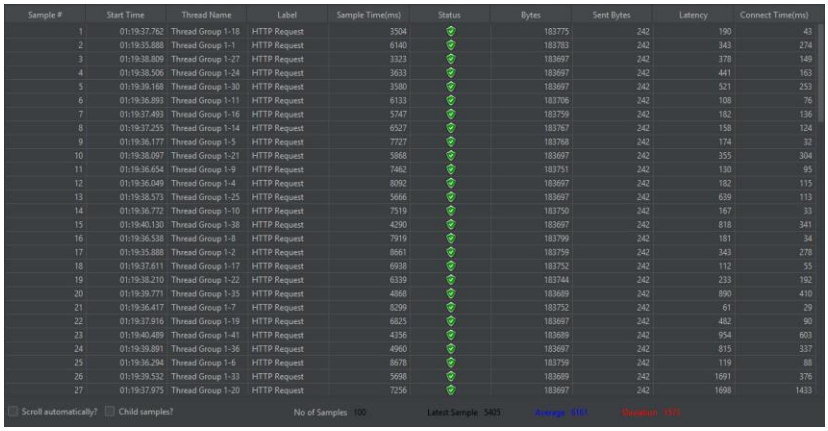

Рисунок 4.6 – Інтерфейс Jmeter при тестуванні навантаженням (рисунок виконано самостійно)

Результати тестування навантаженням приведені на рисунках 4.7, 4.8, 4.9.

Provider	Architecture	Test 2.1 (100 requests)				
		Average Response Time (ms)	CPU Average (%)	Memory Average (Mb)	Latency (ms)	Error Count
AWS	Monolith	503	23	520	104	0
AWS	Microservices Shared DB	598	32	773	132	0
AWS	Microservices Separate DB	656	34	732	140	4
AWS	Zero Trust	703	39	902	204	2
AWS	Encryption	694	32	722	166	0
AWS	Zero Trust + Encryption	863	35	906	232	4
Azure	Monolith	499	22	490	100	0
Azure	Microservices Shared DB	602	24	765	145	1
Azure	Microservices Separate DB	637	31	723	135	0
Azure	Zero Trust	703	36	806	210	3
Azure	Encryption	754	38	843	170	0
Azure	Zero Trust + Encryption	1001	42	902	253	6

Рисунок 4.7 – Результати тесту 2.1 (рисунок виконано самостійно)

Provider	Architecture	Test 2.2 (1000 requests)				
		Average Response Time (ms)	CPU Average (%)	Memory Average (Mb)	Latency (ms)	Error Count
AWS	Monolith	1342	29	990	105	0
AWS	Microservices Shared DB	1678	34	1022	150	1
AWS	Microservices Separate DB	2234	36	1048	140	6
AWS	Zero Trust	2839	46	1055	200	5
AWS	Encryption	2314	37	1034	160	2
AWS	Zero Trust + Encryption	2993	51	1407	250	8
Azure	Monolith	1453	26	1043	132	12
Azure	Microservices Shared DB	1543	28	1023	223	0
Azure	Microservices Separate DB	2122	28	1012	343	0
Azure	Zero Trust	2331	35	1055	211	1
Azure	Encryption	2221	29	1034	213	0
Azure	Zero Trust + Encryption	3124	47	1207	245	2

Рисунок 4.8 -Результати тесту 2.2 (рисунок виконано самостійно)

Provider	Architecture	Test 3.3 (10000 requests)				
		Average Response Time (ms)	CPU Average (%)	Memory Average (Mb)	Latency (ms)	Error Count
AWS	Monolith	3364	78	3243	239	9
AWS	Microservices Shared DB	3976	84	3564	342	6
AWS	Microservices Separate DB	3990	83	3425	245	12
AWS	Zero Trust	5378	92	3453	274	12
AWS	Encryption	3904	85	3224	356	4
AWS	Zero Trust + Encryption	6789	97	3246	453	15
Azure	Monolith	3943	76	3965	236	3
Azure	Microservices Shared DB	5443	79	3432	443	5
Azure	Microservices Separate DB	4005	83	3364	321	8
Azure	Zero Trust	5049	96	3775	321	10
Azure	Encryption	4234	87	3532	235	1
Azure	Zero Trust + Encryption	5506	99	3245	212	13

Рисунок 4.9 – Результати тесту 2.3 (рисунок виконано самостійно)

4.3 Проведення безпекових тестів

Penetration тестування, або тестування на проникнення, є важливою частиною забезпечення безпеки інформаційних систем. Воно включає в себе методичний процес імітації атак на систему з метою виявлення вразливостей, які можуть бути використані зловмисниками для несанкціонованого доступу або викрадення даних. У нашому дослідженні ми провели серію penetration тестів для різних архітектур програмної системи, щоб оцінити їхню стійкість до потенційних атак. Першим кроком у проведенні penetration тестування є підготовка. Це включає в себе визначення цілей тестування, обсягів тестування, методів, які будуть використовуватися, і розробку плану тестування. У нашому випадку ми зосередилися на перевірці основних вразливостей, таких як SQL-ін'єкції, Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), а також тестування на вразливості в авторизації та аутентифікації.

Спочатку розглянемо більш детально деякі види вразливостей.

SQL-ін'єкції (SQLi) є одним з найпоширеніших та найбільш небезпечних типів атак на веб-додатки. Вони виникають, коли зловмисники вводять шкідливий SQL-код у форму введення даних або URL-адресу, який потім виконується базою даних. Це дозволяє зловмисникам маніпулювати запитами до бази даних, отримувати несанкціонований доступ до даних, змінювати або видаляти їх.

Найбільша небезпека SQL-ін'єкцій полягає в тому, що вони можуть обійти механізми автентифікації і авторизації, що дозволяє зловмисникам діяти від імені законного користувача. Для запобігання SQL-ін'єкціям рекомендується використовувати підготовлені запити (prepared statements) з параметризованими запитами, валідацію та санітизацію введених даних, а також застосування ORM (Object-Relational Mapping) інструментів. Інша важлива практика — мінімізація прав доступу користувачів до бази даних, що знижує ризики від успішних атак.

Міжсайтовий скриптинг (XSS) є типом вразливості, яка дозволяє зловмисникам вставляти шкідливі скрипти у веб-сторінки, які переглядають інші користувачі. Це може призвести до крадіжки особистих даних, викрадення сесій або зміни вмісту сторінок. Існує три основні типи XSS-атак: збережений, відображений і DOM-based. Для запобігання XSS важливо перевіряти та екранувати всі вхідні дані та впроваджувати політики безпеки контенту (CSP). Ефективний захист також включає використання безпечних бібліотек і фреймворків.

Cross-Site Request Forgery (CSRF) — це тип атаки, при якій зловмисник змушує користувача виконати небажані дії на веб-сайті, де той вже аутентифікований. Це може призвести до несанкціонованої зміни даних, передачі коштів або інших небезпечних дій без відома користувача. CSRF-атаки використовують довіру веб-сайту до браузера користувача. Захист від CSRF включає використання токенів, які підтверджують справжність запиту, перевірку реферера та впровадження механізму перевірки походження запиту. Інші методи захисту включають використання подвійної передачі кукі та впровадження захисних заголовків, таких як SameSite.

Було протестовано кожну версію архітектур.

Тестування монолітної архітектури показало наявність кількох критичних вразливостей. Виявлено можливість виконання SQL-ін'єкцій через параметри запитів, що дозволяло зловмисникам отримати несанкціонований доступ до бази даних. Було успішно виконано декілька SQL-ін'єкцій, що дозволили витягти конфіденційні дані. Також виявлено кілька місць, де можливо здійснити XSS-

атаки, що дозволяють вставляти шкідливий скрипт у відповіді сервера, який виконується на стороні клієнта. Недосконалі механізми аутентифікації дозволяли здійснити brute force атаки для отримання паролів користувачів.

При тестуванні мікросервісної архітектури з використанням спільної бази даних було виявлено, що незважаючи на розділення на мікросервіси, вразливість до SQL-ін'єкцій зберігається, хоча й локалізована до конкретного сервісу. Відсутність захисту від CSRF-атак дозволяла здійснювати несанкціоновані дії від імені користувача.

При тестуванні мікросервісної архітектури з використанням роздільних баз даних з’ясувалося, що таким чином вдалося підвищити рівень безпеки, але деякі проблеми залишилися. Вразливості в механізмах асинхронної взаємодії між сервісами дозволяли здійснювати атаки на інші сервіси через незахищені черги повідомлень. Неправильна конфігурація безпеки: Деякі сервіси мали неправильні налаштування доступу до баз даних, що дозволяло зловмисникам отримати доступ до даних.

Впровадження принципу Zero Trust значно підвищило рівень безпеки. Застосування багатофакторної аутентифікації та детальна перевірка прав доступу знизили ризик несанкціонованого доступу. Виявлення підозрілої активності та своєчасне реагування дозволило знизити ризики експлуатації вразливостей.

Комбінація Zero Trust та шифрування показала найвищий рівень безпеки:

Поєднання принципів Zero Trust та шифрування даних забезпечило високий рівень захисту від більшості типів атак. Незважаючи на високу захищеність, тестування виявило деякі менш значущі вразливості, пов’язані з неправильними конфігураціями та людським фактором.

Penetration тестування показало, що застосування сучасних архітектурних підходів і принципів безпеки, таких як Zero Trust та шифрування, значно підвищує рівень безпеки програмної системи. Монолітна архітектура виявилася найбільш вразливою, тоді як мікросервісні архітектури з окремими базами даних і впровадженням Zero Trust та шифрування забезпечили найвищий рівень захисту від потенційних атак.

ВИСНОВКИ

У ході дослідження було проаналізовано шість різних архітектурних підходів до створення програмної системи, яка працює з конференційними даними. Використовуючи спроектовані архітектури було створено шість різних версій програмної системи з однаковою функціональністю. Кожну версію системи було розгорнуто в інфраструктурі AWS та Microsoft Azure та піддано різним видам тестування задля перевірки впливу тих чи інших архітектурних рішень на безпеку даних та продуктивність системи.

У ході дослідження ми створили і протестували різні версії програмної системи з використанням різних архітектурних підходів, таких як монолітна архітектура, мікросервісна архітектура з спільною базою даних, мікросервісна архітектура з роздільними базами даних та асинхронною взаємодією між сервісами, покращена архітектура з використанням принципу Zero Trust, застосування шифрування даних в спокої та при передачі, а також комбінація Zero Trust та шифрування.

Монолітна архітектура показала найнижчі показники середнього часу відповіді та споживання ресурсів, що свідчить про її ефективність в умовах невеликих навантажень. Проте, при зростанні складності та масштабів системи, монолітна архітектура стає менш гнучкою і важкою в обслуговуванні.

Мікросервісна архітектура з спільною базою даних забезпечила кращу масштабованість і гнучкість порівняно з монолітом, проте спільне використання бази даних декількома сервісами створює певні виклики для продуктивності та узгодженості даних.

Мікросервісна архітектура з роздільними базами даних та асинхронною взаємодією між сервісами забезпечила кращу ізоляцію сервісів та зменшила навантаження на бази даних. Це дозволило покращити масштабованість та надійність системи, проте додаткові витрати на управління та координацію між сервісами підвищили складність системи.

Використання принципу Zero Trust значно підвищило безпеку системи, але призвело до збільшення середнього часу відповіді та споживання ресурсів. Zero Trust моделі забезпечують більш строгий контроль доступу та постійну перевірку кожного запиту, що ускладнює можливості несанкціонованого доступу.

Застосування шифрування даних в спокої та при передачі також покращило безпеку системи, зменшивши ризик компрометації даних у випадку їх перехоплення або витоку. Однак, це призвело до певного зростання середнього часу відповіді через додаткові операції шифрування та дешифрування даних.

Комбінація Zero Trust та шифрування забезпечила найвищий рівень безпеки, але також призвела до найбільшого зростання середнього часу відповіді та споживання ресурсів. Цей підхід є найбільш підходящим для систем, де безпека даних є критично важливою, навіть за умови компромісу з продуктивністю.

Загалом, вибір архітектурного підходу залежить від конкретних вимог до системи, включаючи масштабованість, гнучкість, продуктивність та безпеку. Монолітна архітектура підходить для невеликих проектів з низькими вимогами до масштабованості. Мікросервісна архітектура є кращою для великих та складних систем, що вимагають високої масштабованості та гнучкості.

Принцип Zero Trust та шифрування даних є ключовими для систем, де безпека є критично важливою. Вони забезпечують високий рівень захисту, але вимагають додаткових ресурсів і можуть знижувати продуктивність системи. Рекомендується використовувати ці підходи в комбінації для досягнення балансу між безпекою та продуктивністю.

Таким чином, результати демонструють, що інтеграція різних архітектурних підходів та принципів безпеки може значно вплинути на продуктивність, масштабованість та безпеку системи. Ретельний аналіз та тестування дозволяють вибрати оптимальний підхід для конкретних умов та вимог проекту.

СПИСОК ДЖЕРЕЛ ПОСИЛАННЯ

1. Carey P. Data Protection: A Practical Guide to UK and EU law / Peter Carey., 2018. – 410 c.

2. Robichau B. Healthcare Information Privacy and Security: Regulatory Compliance and Data Security in the Age of Electronic Health Records / Bernard Peter Robichau., 2014. – 210 c.

3. Health Information Privacy. [Електронний ресурс] // U.S. Department of Health & Human Services. (n.d.). – Режим доступу до ресурсу: https://www.hhs.gov/hipaa/index.html (дата звернення 14.05.2024).

4. Whitman M. E. Principles of Information Security. Cengage Learning. / M. E. Whitman, H. J. Mattord., 2011. – 235 c.

5. Natarajan N., Ranjani S. S. Two Factor Authentication Techniques in Cloud Computing: A Survey // International Journal of Computer Applications. – 2020. – Vol. 173.

6. What is CIA Triad of Information Security. [Електронний ресурс] – Режим доступу до ресурсу: https://aptien.com/en/kb/articles/what-is-cia-triad (дата звернення 14.05.2024).

7. Schneier B. Applied Cryptography: Protocols, Algorithms, and Source Code in C / Bruce Schneier., 1995. P. 389–408.

8. Mather T. Cloud Security and Privacy: An Enterprise Perspective on Risks and Compliance / T. Mather, S. Kumaraswamy, S. Latif., 2009. P. 264-298.

9. Whitman M. E. Principles of Information Security. Cengage Learning. / M. E. Whitman, H. J. Mattord., 2011. – 235 c.

10. Natarajan N., Ranjani S. S. Two Factor Authentication Techniques in Cloud Computing: A Survey // International Journal of Computer Applications. – 2020. – Vol. 173, Issue 1. – P. 1-7. DOI: 10.5120/ijca2020921566.

11. Al-Fakhri S., Drusche D. Cryptography-Based Data Security in Cloud Computing: A Review // IEEE Access. – 2019. – Vol. 7. – P. 17029-17042. DOI: 10.1109/ACCESS.2019.2894804.

12. Gilman E. Zero Trust Networks: Building Secure Systems in Untrusted Networks / E. Gilman, D. Barth., 217. – 238 c.

13. Encryption in-transit and Encryption at-rest - Definitions and Best Practices. [Електронний ресурс] – Режим доступу до ресурсу: https://www.linkedin.com/pulse/encryption-in-transit-at-rest-definitions-best-valerio-de-sanctis/ (дата звернення 17.05.2024).

14. Kachko O., N. Bilous , Semerkov V. Research on methods for secure web applications development Information Technologies in Innovation Business (ITIB) 7-9 October, 2015, Kharkiv, Ukraine Proceedings of ITIB, p.26-27

15. Cambell E. Microservices Architecture: Make the architecture of a software as simple as possible / Edward Cambell., 2016. – 96 c.

16. Gilbert K. Hands-On AWS Penetration Testing with Kali Linux: Set up a virtual lab and pentest major AWS services, including EC2, S3, Lambda, and CloudFormation / Gilbert., 2019.

17. Klaffenbach F. Implementing Azure Solutions: Eliminate the pain point of implementation / Florian Klaffenbach., 2017.

18. Docker Documentation. Security. [Online]. Available: https://docs.docker.com/engine/security/

19. Turnbull J. The Docker Book: Containerization Is the New Virtualization / James Turnbull., 2014.

20. MongoDB Documentation. [Електронний ресурс] // MongoDB. (n.d.). – Режим доступу до ресурсу: https://docs.mongodb.com/ (дата звернення 19.05.2024).

21. AWS Documentation. Security, Identity & Compliance. [Online]. Available: https://docs.aws.amazon.com/security/

22. Jepsen, K. Call me maybe: Amazon DynamoDB. [Електронний ресурс]. // Kyle Kingsbury. – 2018. – Режим доступу до ресурсу: https://jepsen.io/analyses/amazon-dynamodb (дата звернення 22.05.2024).

23. Azure Security Center Documentation. [Електронний ресурс] // Microsoft Azure. (n.d.). – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/azure/security-center/ (дата звернення 01.06.2024).

24. Cloud Security Best Practices: Your Definitive Guide to Securing Cloud Environments / J. Smith, M. Johnson. – Wiley, 2019.

25. Радзівіл В. В., Демченко В. В. Архітектура захищених мікросервісів у хмарному середовищі // Інформаційні технології та комп'ютерна інженерія. – 2018. – № 2 (54). – С. 58-70.

26. Microsoft Azure Documentation. Security Center. [Online]. Available: https://docs.microsoft.com/en-us/azure/security-center/

27. Sharma R., Jain N. A Comprehensive Study on Zero Trust Model for Security Architecture // International Journal of Computer Science and Information Security. – 2019. – Vol. 17, Issue 5. – P. 34-39.

28. I. Afanasieva, N. Golian, O. Hnatenko, Y. Daniiel, K. Onyshchenko. Data exchange model in the internet of things concept // Telecommunications and Radio Engineering (English translation of Elektrosvyaz and Radiotekhnika), 2019, 78(10), p. 869-878