Міністерство освіти і науки України

Харківський національний університет імені В. Н. Каразіна

Факультет комп'ютерних наук

Кафедра теоретичної та прикладної системотехніки

«Затверджую»
Зав. кафедри теоретичної та
прикладної системотехніки
_____ д.т.н., проф. С. І. Шматков
«___» _____ 2021 р

Пояснювальна записка

до кваліфікаційної роботи
магістра

на тему: «**МОДЕЛЬ ПРОГРАМНОГО КОМПЛЕКСУ УПРАВЛІННЯ РОЗРОБКОЮ КЛІЄНТ-СЕРВІСНОЇ СИСТЕМИ**»

Виконав:

студент 6 курсу, групи КУ– 61
Галузь знань: 15 – Автоматизація та приладобудування
за спеціальністю 151 – Автоматизація та комп'ютерно-інтегровані технології.
БІЛЬСЬКИЙ Георгій Миколайович

_____

Керівник: д.т.н., доцент
ЛАБЕНКО Дмитро Петрович

_____

Рецензент: …
…

_____

Захищено на засіданні
Атестаційної комісії № 39
протокол № __ від __.12.2021 р.
Оцінка _____ / _____
Голова Атестаційної комісії
д.т.н., професор
_____**МІНУХІН**
Сергій Володимирович

Харків – 2021

АНОТАЦІЯ

Пояснювальна записка до кваліфікаційної роботи магістра складається зі вступу, трьох розділів, висновків, списку використаних джерел і семи додатків. Загальний обсяг роботи складає 155 сторінок, із яких 115 сторінок основної частини. Робота містить 20 рисунків, 5 таблиць, та 39 найменувань списку використаних джерел на 3 сторінках.

Об'єктом дослідження є комплекс програмного забезпечення для управління процесами проектування, розробки та підтримки програмного продукту.

Предметом дослідження є взаємодія з різними типами інтерфейсів (графічний та консольний), розділення прав доступу користувачів, веб-додатки SPA, додатки для ПК та мобільні платформи Android та iOS, оркестрація контейнерів у середовищі Docker, автоматизація процесів управління інфраструктурою, серверне апаратне забезпечення на базі архітектури ARM64.

Метою роботи є централізація процесів проектування, розробки та підтримки програмних продуктів, забезпечення інтеграції різних типів інструментального та сервісного ПЗ у єдину систему, що є незалежною від зовнішнього середовища та підтримує роботу при обмежених системних ресурсах, та автоматизація процесів пов'язаних з збіркою, тестуванням та розгортанням ПЗ продуктів.

Створення універсальних засобів, що включають у себе уніфікований функціонал багатьох інструментів є критичним для створення комплексного клієнт-сервісного ПЗ та скорочення часу на налаштування та адаптацію процесів та інструментів до різних цільових технологічних баз продуктів.

Ключові слова: *аккаунт, сервіс, система управління проектами, додаток, веб-портал, 2FA, розділений доступ до ресурсів, інтеграція, розширення, єдина аутентифікація, клієнтські бібліотеки, схема Protobuf, модель даних, розгортання сервісів.*

ABSTRACT

The explanatory note for the masters qualification work consists of an introduction, three main sections, conclusions, list of information sources and seven addendums. Total work amount contains 155 pages, from which there are 115 pages of the main section. Work contains 20 illustrations, 5 tables, 4 addenums and 39 names of the information sources list on 3 pages.

The object of the study is a software complex for design, development and support process management of a software product.

The subject of the research is interaction with different user interface types (graphical and console), separation of user access rights, single-page web applications, applications for PC and mobile platforms Android and iOS, container orchestration in the Docker environment, infrastructure control process automation, server hardware based on ARM64 architecture.

The aim of the work is to centralize design, engineering and software product support processes, implementation of instrument and service software integration into a single system, which is independent from external environment and supports operation with limited system resources, and process automation for software build, testing and deployment.

Creation of universal utilities, which include unified functionality of multiple instruments is crucial for creation of complex client-service software and decreasing time spent on setup and adaptation of processes and instruments to different target products' technology foundations.

Key words: account, service, project management system, application, web portal, 2FA, separate resource access, integration, extension, single sign on, client libraries, Protobuf schema, data model, service deployment.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

IAM	–	Identity & Account Management (англ.) – система управління особистостями та обліковими записами.
IoT	–	Internet of Things (англ.) “Інтернет речей”, технологічна система автоматизації рутинних операцій.
ПЗ	–	програмне забезпечення.
ПК	–	персональний комп’ютер.
DB/БД	–	Database (англ.) База даних.
Аккаунт	–	Обліковий запис користувача у системі.
HTTP	–	HyperText Transfer Protocol (англ.) – протокол передачі гіпертексту.
REST	–	Representational State Transfer (англ.) – передача стану представлення.
UI	–	User Interface (англ.) – інтерфейс користувача.
Authentication Authorization registration	–	аутентифікація/авторизація/реєстрація користувача у системі.
CRUD		Create, Read, Update, Delete (англ.) – створення, читання, оновлення, видалення даних.
ORM		Object-Relational Mapping (англ.) – об’єктно-реляційне з’єднання.
HTML		HyperText Markup Language (англ.) – мова маркування гіпертексту
CSS		Cascading Style Sheets (англ.) – каскадні таблиці стилю.
API		Application Programming Interface (англ.) – інтерфейс розробки програми.
Pay-per-use		плата лише за використаний ресурс.
PDMS		Project Definition & Management Service (англ.) – сервіс визначення та керування проектами.
OPS		Operations Management Service (англ.) – сервіс керування операційною діяльністю проектів.
IDE		Integrated Development Environment (англ.) – інтегроване середовище розробки ПЗ.
Frontend		сторона взаємодії з користувачем, застосування UI.
Backend		сторона логіки, обслуговування запитів від сторони Frontend та роботи сервісів.
Microservices		архітектура ПЗ з дуже низьким рівнем сполучення та залежностей, використована для виконання окремих завдань.
Self-hosted		самостйно розміщений ресурс, використовуючи власну інфраструктуру (сервіс, веб-сайт, тощо).
SPA		Single-page application (англ.) – односторінковий додаток для веб-платформи (працює як односторінковий динамічний веб-сайт).

2FA	Two-factor authentication (англ.) – двохфакторна аутентифікація.
RBAC	Role-based access control (англ.) – контроль доступу за ролями.
Single Sign On	єдина аутентифікація у обліковий запис для різних сервісів та клієнтів.
Data Warehouse	система збереження та управління даними.
CI та CD	Continuous Integration та Continuous Deployment (англ.) – неперервна інтеграція та розгортання продукту у цільовому середовищі.
IPC	Inter-process communication (англ.) – комунікація, обмін даними або інша взаємодія між програмними елементами.
SaaS	Service-as-a-Service (англ.) – метод надання послуг типу “сервіс-як-сервіс”, по якому користувачу надається апріорі налаштований та готовий для використання сервіс.
HPC	High Performance Computing (англ.) – високопродуктивні обчислення.
OIDC	OpenID Connect (англ.) – стандарт багатоклієнтської авторизації та аутентифікації користувачів.
GPU	Graphics Processing Unit (англ.) – елемент обробки графіки, відеокарта, у контексті апаратного забезпечення ПК.
JSON	JavaScript Object Notation (англ.) – мова опису елементів для JavaScript (насправді не тільки для JavaScript, назва є історичною).
ACID	Atomicity, Consistency, Isolation, Durability (англ.) – набір властивостей, що є необхідним для гарантії надійної роботи транзакційної БД.
URL, або URI	Universal Resource Link, Universal Resource Identifier (англ.) – універсальний ідентифікатор ресурсу.
MVVM	Model-View-ViewModel (англ.) – принцип проектування архітектури інтерактивних додатків з розділенням рівнів логіки (представлення, даних та інших).

ВСТУП

Концепція клієнт-серверної взаємодії була започаткована ще при використанні перших багатокористувацьких цифрових систем, такі як термінальні сервери, для виконання об'єднаним чином складних завдань або доступу до спільних ресурсів. У часи 60-х років минулого століття головним використанням подібних машин виробництва IBM було виконання бухгалтерських завдань, наукових обчислень та інших завдань, які в основному біли орієнтовані на внутрішні потреби окремих організацій. Використання подібного аппаратного забезпечення потребувало складних та дорогих процесів обслуговування та наявності тренованих операторів. Одним з головних обмежень для збільшення популярності та застосування у більш розповсюджених та простих завданнях також була відсутність глобальної цифрової мережі, що могла б поєднувати мільярди людей по всьому світу, що є присутньою на даний час та має зростаючу популярність [1].

За час технологічного та індустріального прогресу було відкрито не лише багато нових методів вирішення існуючих проблем, а й багато нових проблем та викликів, пов'язаних як з полегшенням життя різних груп людей, так і з вирішенням глобальних проблем, у тому числі безпекових. Основополягаючим у створенні нових рішень та методів використання програмних та апаратних технологій став розвиток архітектурних підходів до будування систем, включаючи не лише інтерфейсну (клієнтську) частину, з якою взаємодіє користувач, а й обслуговуючу, систему логіки, що займається обробкою даних та забезпечує обмін інформацією серед великої кількості користувачів.

Традиційні методи та підходи до проектування та будування рішень подібного типу включають у себе як використання примітивних (однак ефективних по даний час) інструментів, такі як ручка та папір, застосування монолітних архітектур навіть у складних за функціональністю системах, обмеження документування програмного коду, специфіки роботи пов'язаних

частин систем, обмежене або навіть виключення використання систем контролю версій, аудиту, захищеного доступу, та інших, що можуть спростити та поліпшити умови та продуктивність роботи над кінцевим продуктом, та інші. Загалом, така тенденція спостерігалась у період зародження хмарних сервісів та популяризації онлайн-середовищ та інструментів спільної роботи, що є невід'ємною основою роботи над складними програмними проектами на даний час.

Розвиток різноманіття послуг та інструментів, що є наявними у цифровому вигляді спонукнуло індустрію на технологічні інновації та вироблення сервісів та інструментів, що у свою чергу слугують для підтримки цільових програмних продуктів, доступних користувачам. Одною з головних змін у порівнянні з застосуванням технологій у минулому столітті – перехід фокусу від вирішення внутрішніх, спеціалізованих та зазвичай одиничних проблем до комплексних рішень, що можуть включати у себе не лише окремий функціонал одного сервісу, а й надавати можливість включати багато інших, надаючи розробникам інструменти для вбудови їх рішень у продукт, що надається одним виробником. Спеціалізація інструментів та сервісів також зазнала розширення, складаючи десятки категорій для застосування при вирішенні різноманітних завдань та проблем. Це є як і окремі засоби, що існують лише всередині компаній для окремих застосувань, так і комплексні пакети програмного забезпечення, що надаються великими компаніями, призначені для будування продуктів будь-якого спектру спеціалізації.

Але з прибуттям все більшої кількості категорій програмних інструментів для проектування та розробки ПЗ виникли також і недоліки застосування таких рішень, що є особливою проблемою для забезпечення незалежності роботи від зовнішніх факторів, та для організацій з обмеженим бдюжетом, що не можуть собі дозволити витрачати великі суми за публічні ресурси корпорацій для надання послуг великій кількості користувачів. Питання безпеки, коштів, незалежності та спеціалізації є одними з основних, що постають перед будуванням самодостатніх, та, у свою чергу, комплексних

продуктів, які використовують інтегровані та спеціалізовані сервіси, не потребуючих проектування та адаптації для кожного нового рішення або ітерації продукту.

Головним викликом при побудові єдиного комплексу, або системи програмного забезпечення, яке буде забезпечувати вирішення завдань систематизованого проектування, розробки та випуску продукту є баланс між інтеграцією та незалежністю компонентів, тому що при відсутності інтеграції будуть відсутні можливості взаємодії між інструментами управління програмним кодом та збірки кінцевої програми, що може застосовувати спеціалізовані конфігурації. У іншому випадку, при відсутності залежності сервісу, що надає доступ до таких конфігурацій, від системи аутентифікації є ризик несанкціонованого доступу та витоку інформації, що може спричинити порушення функціонування кінцевих продуктів та виток персональних даних користувачів, або інші негативні наслідки. Застосування єдиних, стандартизованих засобів надає можливість полегшити розробникам процес будування продукту та автоматизувати багато операцій, що зазвичай довелося би виконувати вручну, такі як публікація оновленої версії програми, або розгортання нового серверу для підтримки стабільної роботи під час зростаючого навантаження від запитів користувачів. У той же самий час, створення окремої, самодостатньої системи, дозволить як зекономити кошти, витрачені за оплату планів використання публічних (а також негарантованих 100% часу) ресурсів, а й забезпечити можливості щодо внесення змін та розширення функціоналу відповідно до довільних вимог, що є неможливим у інших варіантах, або є дуже обмеженим.

У даній науковій роботі представлено дослідження, спрямовані на аналіз загальної проблематики використання спеціалізованого програмного забпзечення при будуванні комплексних програмних рішень, огляд особливостей різних категорій наявних систем та окремих програмних інструментів, включаючи публічно-доступні, їх переваги та недоліки. За результатами досліджень та постановкою обгрунтувань подальшої роботи

спроектована модель спеціалізованого комплексу ПЗ, що направлений на усунення недоліків наявних інструментів та підходів до їх використання, у свою чергу надаючи спільні методи та засоби вирішення проблеми побудови програмних продуктів, у свою чергу спираючись в основному на об'єднання різних ступенів інтеграції, а також обмеження для підтримки основних вимог щодо фінансової доступності та самодостатності для застосування за призначенням у організаціях різних типів, або навіть у окремих проектах індивідуальної розробки.

Архітектура спроектованої моделі є модульною, розділеною на основні та додаткові компоненти, що не є необхідними для роботи системи в основі. Ці додаткові компоненти є одною з частин новизни проекту, через використання моделі розширюваності, можливості додавання та вилучення функціоналу без зміни стану або конфігурації основної системи у цілому.

Представлена робота має значення через висвітлення проблем залежності багатьох рішень, якими користуються мільйони людей у світі, від лише декількох сервісів корпорацій, що регулюють як правила використання, так і цінову політику, при цьому маючи проблеми з підтримкою користувачів, а також проблеми витрат часу та ресурсів через відсутність централізованих систем та інструментів для розробників для роботи над проектами, використовуючи однаковий функціонал, сервіси та інструменти незалежно від платформи, проекту або пристрою.

Модель програмного комплексу сервісів та інструментів, спроектована за результатами проведеної роботи включає у себе декілька основних компонентів, а також можливість впровадження нового функціоналу та підтримки різноманітних типів програмних засобів, що застосовуються для проектування, розробки та підтримки програмних продуктів.

Головною метою практичної частини роботи є проектування та розробка моделі інтегрованого комплексу програмного забезпечення, що буде відповідати вимогам мінімальних витрат бюджету, забезпечувати роботу якомога більшої кількості сервісів та інструментів при мінімальних системних

вимогах, включати у себе підтримку динамічної інтеграції між різними типами таких сервісів, а також представляти собою спільний візуальний інтерфейс для користувачів, об’єднуючи спільні можливості у єдиному виді та функціональності.

РОЗДІЛ 1.
ТЕОРЕТИЧНИЙ ОГЛЯД ПРИЧИН ТА ПРОБЛЕМАТИКИ СИСТЕМАТИЗАЦІЇ УПРАВЛІННЯ ПРОЦЕСАМИ РОЗРОБКИ КЛІЄНТ-СЕРВІСНИХ ПРОГРАМНИХ СИСТЕМ

Розуміння причин, цілей, та складових методів систематизованого контролю за процесом проектування, розробки та підтримки комплексних програмних систем є основою для встановлення проблематики та основних питань для проведення досліджень необхідності проектування нового рішення. Основним сподвигом до роботи над новою системою є виявлені недоліки, усунення яких допоможе оптимізувати процесс, або необхідність побудови нової архітектурної моделі для програмних продуктів, нових підходів до розробки ПЗ такого типу.

Для отримання розуміння цільової предметної області потрібно провести огляд та встановити основні підходи до процесів та складові розробки програмних продуктів, та архітектурні моделі такого програмного забезпечення. Основні моменти, на які звертається увага при огляді проблематики управління процесами розробки комплексного ПЗ та цільових архітектур:

1. Суть архітектурної моделі.
2. Причини вибору цільової моделі та її типове застосування.
3. Основні принципи проектування ПЗ на основі обраної архітектурної моделі та елементи, що є основними складовими моделі, їх зв’язок.
4. Підходи до процесу розробки ПЗ на базі обраної архітектури у контексті робочого процесу (застосування ресурсів, цикл розробки та ін.).
5. Основні особливості архітектурної моделі та відмінності від інших (як у практичному, так і суттевому плані).

Одним з найважливіших питань огляду є інтеграція інструментарію розробки у процес, його катеоргії та типове застосування. Відповіді на ці питання є основою для розуміння наявних проблем та можливих методів їх вирішення, або усунення у цілому.

Для визначення основних відмінностей архітектурних схем, у тому числі можливих недоліків та переваг, а також порівняння обраних для огляду моделей з іншими у наявності встановлюються наступні критерії:

- Канал зв'язку – визначає підтримувані типи/платформи, на базі яких встановлюється канал взаємодії між кінечними компонентами архітектурної моделі;
- Максимальна кількість користувачів/клієнтів;
- Джерело авторитативної інформації – визначає компонент, інформація з якого не підлягає додатковим перевіркам на достовірність та цілісність, та вважається достовірною апріорно;
- Відносна складність реалізації і підтримки – визначення відносно інших порівнюваємих архітектур складових для прийняття до уваги, що потребуються для створення мінімально життєздатного програмного продукту (MVP) на її основі;
- Відносні фінансові витрати – визначення відносно інших порівнюваємих архітектур фінансових витрат на матеріальні та організаційні ресурси, що потребуються для створення мінімально життєздатного програмного продукту (MVP) на її основі;
- Відносні часові витрати – визначення відносно інших порівнюваємих архітектур витрат часу для проектування та розробку, що потребуються для створення мінімально життєздатного програмного продукту (MVP) на її основі;
- Особливості – додаткові складові, що є унікальними для кожної окремої архітектурної моделі.

1.1 Клієнт-серверні рішення та принципи розробки.

Архітектура клієнт-сервер є основою для будь-якого програмного забезпечення, що розраховано на більш ніж одного користувача. Вона включає у себе три основні компоненти, багато можливих варіантів їх розміщення (включаючи різні топології зв'язків) та застосувань на практиці, через можливість використання у ролі клієнта не лише користувача-людини.

1.1.1 Причини та цілі застосування.

Будування програмного та апаратного забезпечення на основі клієнт-серверної моделі є найпоширенішим підходом до створення систем, що у більшості ситуацій розраховані на забезпечення спільного доступу до ресурсів, або надання послуг більш ніж одному користувачу в одну одиницю часу.

Серед можливих причин вибору такої моделі при проектуванні програмних продуктів можна виділити наступні:

- Необхідність підтримки великої кількості користувачів;
- Необхідність забезпечення контролльованого доступу до інформації;
- Необхідність централізованого збереження інформації, без надлишкового обміну між клієнтами та займанням більшого місця у сховищі;
- Необхідність у більшій кількості апаратних ресурсів, що недоступні на клієнтських пристроях окремо.

Причини застосування даного типу архітектурної моделі можуть бути як об'єднаними, так і незалежними, залежно від вимог до системи, що базуються на кінцевих цілях її застосування. Серед типових прикладів впровадження систем з даною архітектурою можна виділити наступні:

- Централізоване сховище спільних ресурсів (даних);

- Система аутентифікації до захищених ресурсів на базі облікових записів;
- Доступ до спільних принтерів та іншого обладнання, що не є безпосередньо під'єднаним до клієнтсьих машин;
- Багаторівневе програмне забезпечення, у якому графічний інтерфейс та пов'язаний код представлено як "клієнт", що використовує вбудований "сервер", який надає API для направлення запитів до бізнес-логіки.

Особливість останнього прикладу складається у виключенні розділення меж компонентів у апаратному забезпеченні, тобто вся система може працювати на одному пристрої, а обмін інформації між клієнтом та сервером відбувається через вбудовану локальну мережу, що надається платформою ОС (localhost). Такий підхід представляє собою розділення лише на 2 зони відповідальності (frontend та backend), без урахування розділення частин логіки між собою, спираючись на унікальну функціональність кожного програмного блоку. Більш поглиблене розділення представляє собою мікросервісна архітектура, огляд якої представлено у п. 1.2.

1.1.2 Принципи побудови та архітектура.

Основними складовими архітектурної моделі "клієнт-сервер" становлять 3 взаємозв'язані компоненти: клієнт, сервер, та зв'язуюча їх мережа.

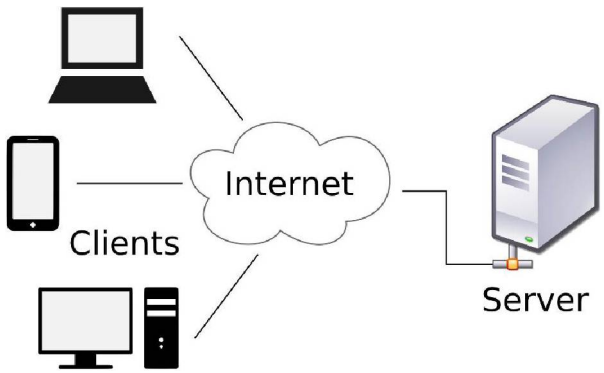

Рисунок 1.1 – архітектурна схема взаємозв'язку компонентів моделі [2]

На типовій схемі, представленій рисунком 1.1, розподілення компонентів представлено як “багато клієнтів на один сервер”, але існують схеми проектування, що вкючають як багатосерверні системи, так і системи з одиничною кількістю кожного типу компонента.

Загалом, типом архітектури “клієнт-сервер” визначаються наступні компоненти та їх можливі кількісні представлення:

- Сервер (один або бальше) – обов’язковий компонент, що може бути представлений як апаратним елементом, так і програмною реалізацією. Представляє собою авторитативну структуру, що може зберігати інформацію, яка є достовірною для всіх клієнтів, або надавати доступ до спільних сервісів та ресурсів.
- Клієнт (один або більше) - обов’язковий компонент, що може бути представлений як апаратним елементом, так і програмною реалізацією, відповідно до вимог до системи та за схожою схемою до сервера. Основним застосуванням є використання ресурсів, що надаються сервером.
- Мережа – зв’язуючий компонент, завдяки якому проводиться взаємодія між клієнтом та сервером. Може бути представлена апаратною (фізичні кабелі з’єднання або бездротова мережа) або програмною реалізацією (за рахунок API операційної системи).

Незалежно від комипонентного розподілення, для побудови систем на базі даної архітектури виділяють три основні рівні взаємодії та логіки:

1. Представлення даних – інтерфейс між користувачем та машиною, забезпечує виведення даних до користувача та прийом команд у відповідь. Даний рівень впроваджується лише на клієнті;
2. Прикладний – реалізує логіка системи, що відповідає за обробку інформації та виконання авторитативних операцій бізнес-логіки;

3. Управління даними – забезпечує контроль зберігання даних та доступу до них.

1.1.3 Основні методи та підходи до розробки.

Вибір архітектурної моделі “клієнт-сервер” для програмного продукту базується на наступних питаннях:

1. Чи є необхідність у забезпеченні доступу до ресурсів та/або сервісів більш ніж одного користувача/клієнтського ПЗ?
2. Чи є необхідність у авторитативному контролі та обробці операцій через неможливість представлення даних у довіреному режимі за допомогою одного пристрою?
3. Чи є необхідність у розподіленому виконанні операцій через нестачу ресурсів на одному клієнтському пристрої або зважаючи на фізичне розмежування користувачів?

Окрема увага потрібна бути приділена визначенню авторитативної схеми, тобто який компонент буде відповідальний за обробку інформації та слугувати достоівірним джерелом. Зазвичай така відповідальність надається серверу, але загалом існують 2 основні схеми:

- Клієнт-авторитативна – основна логіка та обробка даних виконується на клієнті, сервер приймає інформацію без додаткових перевірок, маючи на увазі апріорну достовірність даних та запитів;
- Сервер-авторитативна – логіка та обробка даних виконуються на стороні сервера, включаючи перевірку даних та запитів від клієнтів на цілісність та відповідність заданим у програмному коді допустимим межам. Зі сторони клієнтів приймання даних зі сервера не потребує додаткових перевірок, через їх апріорну достовірність.

Визначення параметрів мережі комунікації між серверними та клієнтськими компонентами є критичним для збереження необхідної швидкості, пропускної можливості та додаткового функціоналу каналів зв'язку. Типові критерії для параметрів компоненту мережі:

- Основа каналу зв'язку: апаратна або програмна (у залежності від розміщення клієнта та серверу);
- Протокол взаємодії:
  - HTTP (для обміну даними),
  - gRPC (для виклику віддалених процедур на клієнті зі сторони серверу та навпаки),
  - TCP (для загального обміну інформацією з контролем отримання),
  - UDP (для загального обміну інформацією без стандартного контролю отримання)

Для багатьох популярних мов програмування існують бібліотеки та API функції, що надають доступ до системних можливостей, які забезпечують реалізацію компонентів архітектури на програмному рівні. Окремо, при використанні апаратних методів, особливо при формуванні мереж, виробники рішень забезпечують пристрої різних категорій з реалізацією різних можливостей управління елементами мережі та правилами її формування та функціонування.

1.1.4 Порівняння з іншими архітектурними моделями.

Загалом, модель "клієнт-сервер" є принципіально відмінною від архітектур єдиного клієнта або peer-2-peer через схему авторитативної обробки інформації.

У схемі peer-2-peer кожна кінцева точка має рівне уявлення про інформацію та спільний доступ до даних. Завдання перевірки достовірності

та цілісності даних покладається на кожний елемент мережі, на відміну від одного сервера у розглядаємій моделі.

Схема єдиного клієнта є найпростішою через покладання всіх вимог щодо роботи за даними та виконання всіх частин логіки на один компонент. Використання такої схеми зазвичай зумовлено вимогами до розробляємого програмного продукту, що може проектуватися як повністю автономне ПЗ.

У таблиці 1.1 наведене більш детальне порівняння трьох схем за основними критеріями.

Таблиця 1.1

Порівняльна таблиця критеріїв архітектур програмних рішень

Критерій	**Єдиний клієнт**	**Peer-2-peer**	**Клієнт-сервер**
Канал зв'язку	-	Апаратний (клієнти на різних пристроях)	Програмний або апаратний (в залежності від розміщення компонентів)
Макс. клієнтів	1	2+	1+
Джерело авторитативної інформації	Клієнт	Клієнт	Клієнт або сервер (в залежності від схеми авторитативності)
Відносна складність розробки продукту	Проста	Складніша	Складна
Відносні фінансові витрати	Низькі	Високі	Найвищі
Відносні часові витрати	Низькі	Великі	Великі

Таблиця 1.1 (продовження)

Критерій	**Єдиний клієнт**	**Peer-2-peer**	**Клієнт-сервер**
Особливості	Всі 3 рівні взаємодії та логіки об'єднані у одному додатку	Не використовує централізоване сховище даних або єдину кінцеву точку для обробки запитів, обмін відбувається між кожним клієнтом	Клієнтські компоненти мають залежність від серверних ресурсів та сервісів для представлення даних та функціоналу користувачам

За результатами порівняння можна встановити, що використання архітектурної моделі клієнт-сервера є найбільш адаптивним рішенням, але є також одним з найскладніших для реалізації. Високі фінансові витрати пов'язані з залученням окремого апаратного забезпечення для серверного ПЗ, а принцип залежності клієнтів від сервера не дозволяє будувати системи з динамічною інфраструктурою у більшості випадків. Однак, у свою чергу застосування архітектури peer-2-peer не надає гарантії єдності та цілісності даних, через можливість дублікації версій на клієнтах з відмінностями, тому для таких ситуацій окремо потребується проектування механізмів підтримки постійності даних на кожному клієнті.

1.2 Мікросервіси.

Архітектура мікросервісів представляє собою еволюцію традиційної моделі клієнт-сервер, що вбачає у собі розділення компонентів на менші, зв'язані між собою логічні елементи, що виконують програмний код, який представляє окремий сервіс з унікальною функціональністю, що не залежить напряму від інших. У свою чергу, зв'язок для взаємодії сервісів є непрямим, тобто відбувається через використання одного зі стандартних протоколів.

Таке розділення забезпечує більш незалежне та стабільне функціонування комплексної системи, в якій є можливість впроваджувати зміни швидше, ніж для традиційної моделі серверу, та забезпечувати більшу стійкість до критичних проблем.

1.2.1 Основні відмінності від традиційних архітектур.

До основних характеристик архітектури мікросервісів можна віднести наступні [3]:

- Незалежні процеси розробки, тестування та підтримки для логічних модулів;
- Низька залежність модулів;
- Можливість незалежного випуску та впровадження частин системи;
- Розділена відповідальність частин системи за окремими розробниками або командами;
- Організація модулів за бізнес-логікою та вимогами до частин системи.

Головною особливістю, у порівнянні з традиційною, клієнт-серверною, є розділення монолітного серверного компоненту на декілька взаємопов'язаних за рівнем, але виконуючих окремі функції, що були запрограмовані для кожного сервісу. Проектування системи, розділеної на окремі логічні блоки забезпечує досягнення наступних переваг перед традиційними монолітними системами:

- Розділення зон відповідальності та залежності при роботі декількох команд або розробників над різними модулями;
- Можливість незалежного масштабування частин системи, згідно з навантаженням та/або іншими метриками, що відносяться до відповідних частин. Така гібридна модель забпзепечує збереження використовуваних ресурсів (у тому числі

процесорних, розмір сховища інформації, оперативної пам'яті) при розрізненому рівні використання різних компонентів системи;

- Прихована реалізація логіки модулів забезпечує швидку ітерацію розробки або значні зміни реалізації, залишаючи незмінним інтерфейс запитів та команд для взаємодії між модулями. Додатково, окремі модулі мають можливість бути реалізованими на базі будь-яких технологій, мов програмування або навіть платформ (апаратних або програмних);
- Більш ефективне застосування ресурсів та індивідуальний контроль за їх використанням через роздільне розміщення модулів системи (як на одному пристрої, так і розподілено на багатьох);
- Використання симетричного принципу взаємодії типу "споживач – виробник";
- Незалежна обробка аварійних станів та вихід з них без залучення до такого стану інших частин системи.

Загалом, впровадження даної архітектурної моделі може привнести до процесу розробки та самих програмних продуктів як переваги, так і притаманні цій моделі недоліки.

Основні переваги вибору мікросервісів:

- Наявність механізму та засобів безперервного впровадженя та розгортання комплексних рішень;
- Покращене обслуговування – кожний сервіс є окремою невеликою частиною систеи, що спрощує його розуміння та внесення змін;
- Покращене тестування та розгортання через незалежність сервісів один від одного;

- Покращене виявлення та ізоляція помилок, особливо пов'язаних з помилками програмної або апаратної платформи, на якій працює сервіс. Порівняно з монолітними рішеннями, у тому числі з архітектурою "клієнт-сервер", падіння одиниці одного типу сервісу не впливає на роботу інших;
- Незалежність від прив'язки програмного коду до одної апаратної або програмної платформи.

Основні недоліки вибору мікросервісів:

- Підвищена складність у розробці через необхідність розробки механізмів IPC для комунікації між сервісами, та обробки окремих помилок та аварійних станів у сервісах. Додаткова складність виникає при проведенні інтегрованого тестування або обміну запитами між багатьма сервісами;
- Підвищена складність при розгортанні та управлінні розподіленою системою подібного типу;
- Підвищені витрати ресурсів порівняно з монолітними рішеннями.

1.2.2 Причини та цілі застосування.

Для розуміння причин вибору даної моделі при побудові комплексних програмних продуктів та кінцевих цілей її впровадження у проектах потрібно поглянути на приклад системи, що побудована на мікросервісній архітектурі. На рисунку 1.2 представлена компонентна схема типового інтернет-магазину.

У даній системі представлені 5 логічних модулів, 2 клієнти та 3 бази даних, що відповідно відносяться до різних рівнів представлення. Розподілення елементів системи по рівням є наступним:

- Представлення даних: мобільний клієнт та веб-браузер, через них відбувається взаємодія користувачів з функціоналом інтернет-магазину та представлення даних у інтерактивному виді;

- Управління даних: БД аккаунтів (Account DB), БД інвентарю/товарів (Inventory DB), БД замовлень (Shipping DB) – включає у себе контроль за збереженням даних у реляційних БД, а також запрограмовані функції для управління даними всередині самих БД;
- Прикладний: розподілений між модулями серверної частини, включаючи сервіси аккаунтів (Account), інвентарю (Inventory), замовлень (Shipping), веб-додатку (Storefront). Дані сервіси виконують основні функції інтернет-магазину, пов’язані з управління особовими записами користувачів, інформації про їх замовлення та контроль персональних даних, а також передачу інформації на запити для висвітлення товарів у додатку або браузері, та застосування списку замовлень для його передачі до постачальників товарів.

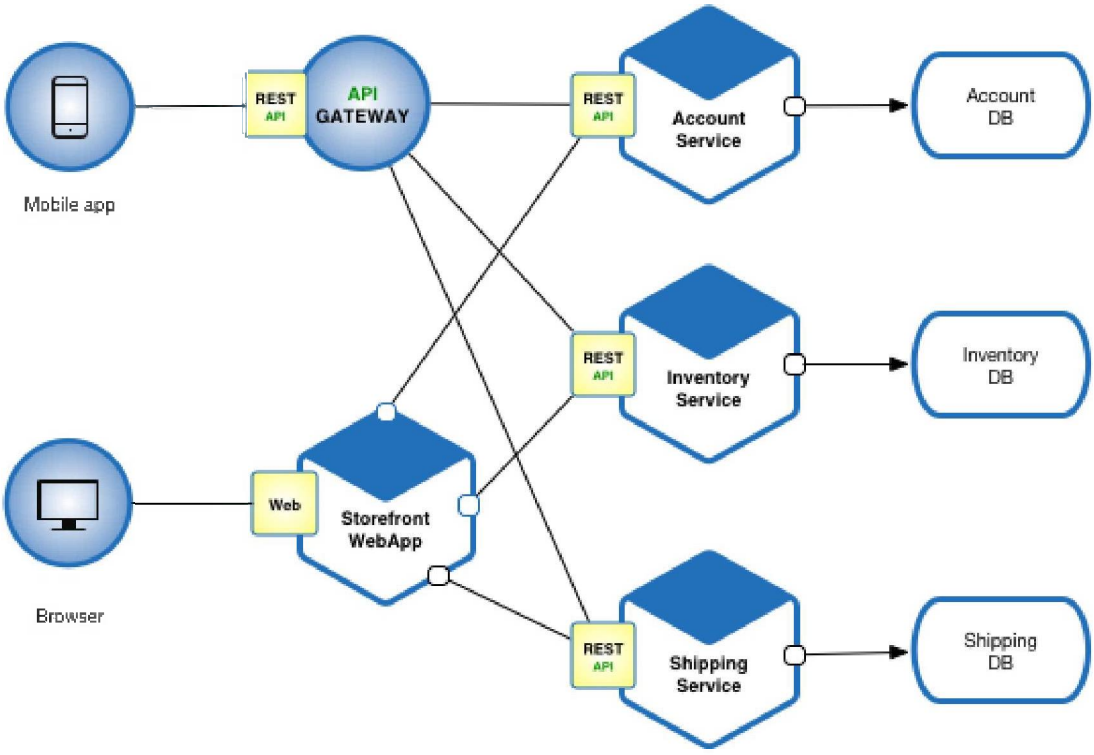

Рисунок 1.2 – компонентна схема типового проекту інтернет-магазину, побудованого на архітектурі мікросервісів

У даному прикладі основними особливостями застосуваннями мікросервісів є наступні:

- Стадартизований протокол комунікації між сервісами, що забезпечує незалежність реалізацій та динамічність у роботі системи загалом, з можливостями розширення або внесення інших змін на серверній частині, що у той же час не має критичного впливу на роботу інших частин системи або взаємодію з нею користувачів;
- Розділене застосування баз даних;
- Єдина точка входу для запитів користувачів з мобільного пристрою до багатьох сервісів, що полегшує обслуговування та стандартизацію форматів даних та інформації про помилки у відповідь; централізоване управління правилами доступу до сервісів, включаючи файрвол.

Загалом, одними з головних чинників при вибору данної архітектурної моделі можуть слугувати наступні (у той самий час, вони не є обов'язковими):

- Робота над програмним продуктом виконується у команді;
- Є необхідним швидке пристосування нових членів комади до середовища роботи над продуктом;
- Цільовий продукт повинен бути адаптивним та мати зрозумілу архітектуру та код;
- Для програмного забезпечення продукту необзідне застосування принципу Continuous Integration (CI) для швидкого розгортання нових версій для користувачів;
- Має пріоритет використання останніх інновацій у бібліотеках, програмних платформах, а також інструметах розробки без внесення постійних архітектурних та програмних змін до проекту загалом;

- Має пріоритет забезпечення автоматичного розширення кількості робочих копій сервісів для забезпечення рівномірного обслуговування запитів та доступності системи у цілому.

1.2.3 Принципи побудови та архітектура.

Для проектування та розробки програмних продуктів з використанням архітектурної моделі мікросервісів важливо встановити відповіді на наступні питання:

- Чи є необхідність у забпезпеченні взаємодії між багатьма користувачами чи набором ресурсів?
- Чи є необхідність у проектуванні роздільних сервісних блоків з унікальною незалежною функціональністю, порівняно з застосуванням монолітної архітектури?
- Чи включено у вимоги до системи підтримка швидкої ітерації змін або підтримка динамічного розширення кількості типів сервісів?
- Чи є критичним забезпеченя розділення даних або частин бізнес-логіки через безпекові або інші причини?

При проектуванні архітектури програмного продукту також важливо мати на увазі певні архітектурні обмеження, які формують суть та переваги використання такої моделі [5]:

1. Еластичність – кожний сервіс повинен мати можливість використовувати як більше, так і менше ресурсів, незалежно від інших сервісів та стану системи у цілому;
2. Стійкість – будь-які перебії або аварійні стани одного сервісу не повинні негативно впливати на інші;
3. Композиція – кожжний сервіс повинен представляти собою одиницю програмного забезпечення, що є унверсальною, та підтримує об’єднання з іншими сервісами у єдину систему;

4. Мінімалізм – кожний сервіс повинен бути спроектований на виконання лише мінімальної кількості завдань, поставлених на нього відповідно до вимог до продукту;
5. Завершеність – у кінечному продукті, кожний сервіс поминен представляти собою завершену програмну одиницю, що повноціно виконує програмну логіку та поставлені на сервіс завдання, забезпечуючи загалом з іншими сервісами стабільність роботи системи у цілому.

З основних компонентів мікросервісної архітектури виділяють три: сам сервіс, або їх сувукупність, якщо йде мова про загальний продукт на базі даної архітектури; метод або протокол взаємодії між сервісами, та принцип управління та контролю за організацією та взаємодією сервісів у рамках єдиної системи.

- Сервіс – один з головних елементів програмного продукту, виконує функціонально відокремлену частиу бізнес-логіки, покладаючись на взаємодію з зовнішнім середовищем та/або з іншими сервісами (якщо встановлено у вимогах до продукту). Може включати у себе відокремлене, внутрішне сховище даних та надавати до нього доступ іншим елементам системи, згідно встановлених правил при проектуванні або конфігурації системи;
- Централізоване управління та контроль сервісів – забезпечує динамічну роботу довільного набору сервісів у рамках єдиної системи, включаючи взаємодію, дотриманя правил обміну даними, збір та обробку службової інформації, а також динамічну зміну стану системи та набору сервісів (за потреби);
- Протокол взаємодії – регулюється елементом управління та слугує єдиним методом комунікації та взаємодії між сервісами. Тип протоколу обирається згідно вимог до системи.

1.2.4 Основні методи та підходи до розробки.

Проектування комплексних програмних продуктів полягає у наступних діях, суть яких – формування представлення функціональності окремих сервісів, їх методів взаємодії, необхідних ресурсів, та схеми відносного розміщення сервісів та інших елементів в рамках системи:

1. Визначення вимог до системи та формування структури необхідних елементів бізнес-логіки;
2. Розподілення елементів бізнес-логіки на незалежні частини, що відносяться до окремих сервісів;
3. Вибір принципу взаємодії між сервісами та іншими ресурсами, що входять до складу кінцевого програмного рішення;
4. Розробка сервісів, інтеграція необхідних ресурсів, формування протоколу взаємодії (API) між сервісами;
5. Проведення індивідуального (unit) та інтегрованого тестування з залученням декількох або усього набору сервісів продукту.

Слідуючи основним принципам філософії будування мікросервісних проектів [6], важливим є і розуміння як оминання так званих “анти-шаблонів впровадження”, які можуть завадити або подовжити розробку кінцевого продукту, привносячи у процес технічний борг або небажані ускладнення. Деякі з головних заблуджень при проектувані з мікросервісною архітектурою [7]:

- Архітектура мікросервісів не є магією та вирішеням всіх проблем при проектуванні одної чи іншої системи;
- Використання такої архітектури не повинно бути кінечною метою при розробці продукту;
- Впровадження мікросервісної моделі повинно бути координованою роботою з усіма задіяними особами у розробці продукту;

- Фокус у впровадженні повинен бути на розділенні бізнес-логіки на окремі сервіси, а не на застосуванні тої чи ішої мікросервісної технології;
- Навіть для комплексних проектів з велиою кількістю логічних елементів повинна матися на увазі розумна межа розділення. При збільшенні кількості сервісів можуть виникати додаткові труднощі та навіть проблеми як у процесі розробки, так і для самого продукту;
- При використанні мікросервісної архітектури потрібно використовувати специфічні методи організації роботи та алгоритми розробки, не використовуючи ті, що можуть бути притаманими традиційним монолітим архітектурам, через можливість виникнення проблем, порушуючих суть та переваги самої цільової архітектури.

Для проектування рішень на основі архітектурної моделі мікросервісів були виділені 6 найбільш популярних паттернів [8]:

1. Aggregator: використовується при будуванні клієнтів, що збирають дані з декількох сервісів для одної операції (наприклад, для веб-сторінки з різними категоріями інформації). При об’єднанні даних з багатьох сервісів, група таких сервісів може бути представлена як єдиний, а індивідуальна кількість може змінюватися у довільному порядку (в залежності від налаштованих параметрів);

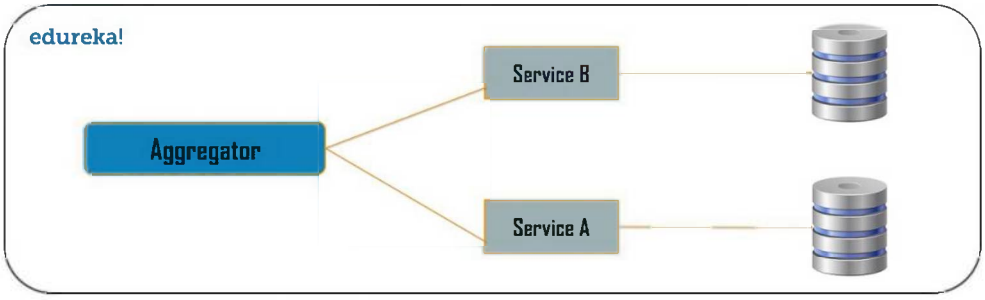

Рисунок 1.3 – типова схема паттерну Aggregator з двома сервісами, об’єднаними в представлення одного через комбіацію окремо отриманмих даних

2. API Gateway: слугує як проміжний шір між клієнтами та сервісами, виконуючи наступні функції:
   a. проксі-сервіс для маршрутизації запитів з однієї публічної вхідної точки до багатьох внутрішніх сервісів, включаючи можливості паттерну Aggregator,
   b. балансування навантаження між різними екземплярами сервісів відповідно до кількості запитів або інших метрик,
   c. забезпечення цілісності та достовірності даних, а також дотримання інших режимів контролю, встановлених у конфігурації, таких як обмеження швидкості запитів або блокування запитів з клієнтів у чорному списку.

   Такий спосіб може бути реалізован як у вигляді єдиної точки входу, так і багатьох, відповідно до вимог розробників продукту (наприклад, забезпечуючи розділення API для мобільних додатків та веб-порталів);

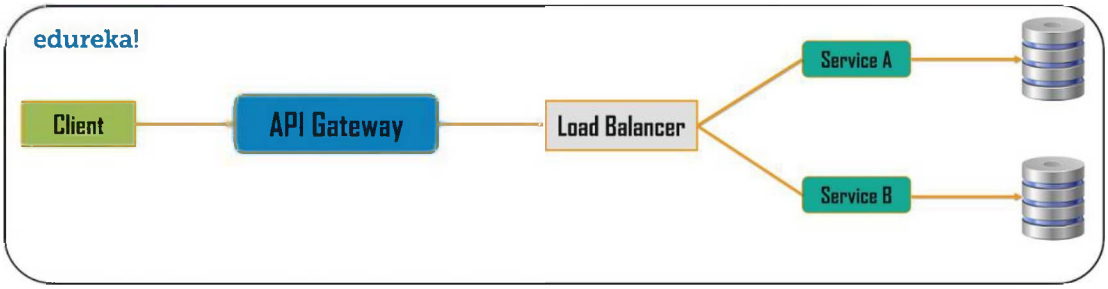

Рисунок 1.4 – типова схема паттерну API Gateway, що включає 2 внутрішні сервіси, єдину точку входу для запитів, та баласувальник навантаження (Load Balancer)

3. Chain of responsibility: менш популярний паттерн, але дуже корисний, коли обробка та фільтрування запитів виконується декількома сервісами, відповідно до їх рівня розміщення. Для таких систем цей паттерн може слугувати для виконання різних операцій, що трансформують самі об’єкти запиту та

відповіді відповідно до формату цільового сервісу (наприклад, для різних протоколів взаємодії між декількома сервісами);

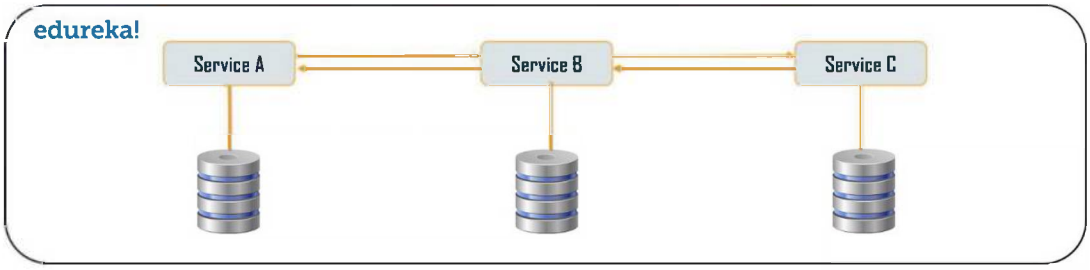

Рисунок 1.5 – типова схема паттерну Chain of responsibility, що включає 3 сервіси, через 2 з яких проходять запити в обов'язковому порядку, піддаючись попередній обробці до потрапляння до сервісу С

4. Async Messaging: слугує для систем, орієнтованих на HPC обчислення, або з дуже великою номінальною пропускною здатністю, зумовленою великою кількістю користувачів. Головною особливістю є пріоритет на мінімальний час від запиту до відповіді та можливість викоання операцій сервісами без очікування результату роботи інших, обробляючих запит (асинхронно);

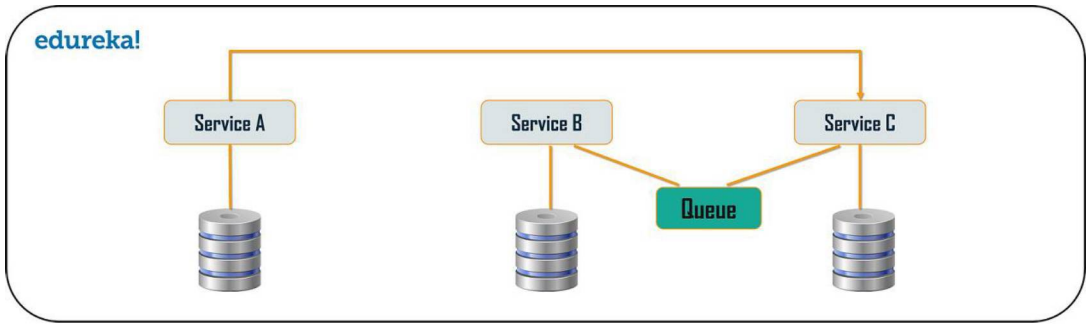

Рисунок 1.6 – типова схема паттерну Async Messaging, що включає групу з 2 синхронних сервісів, та одни асинхронний, який обробляє запит незалежно від группи. Синхронність обробки у прикладі зумовлена наявністю черги запитів

5. Shared Data: слугує для надання декільком сервісам одночасно доступ до єдиної копії даних, що насамперед вирішує проблему дублювання даних між різними сервісами, а також зекономлює ресурси через застосування спільного місця зберігання;

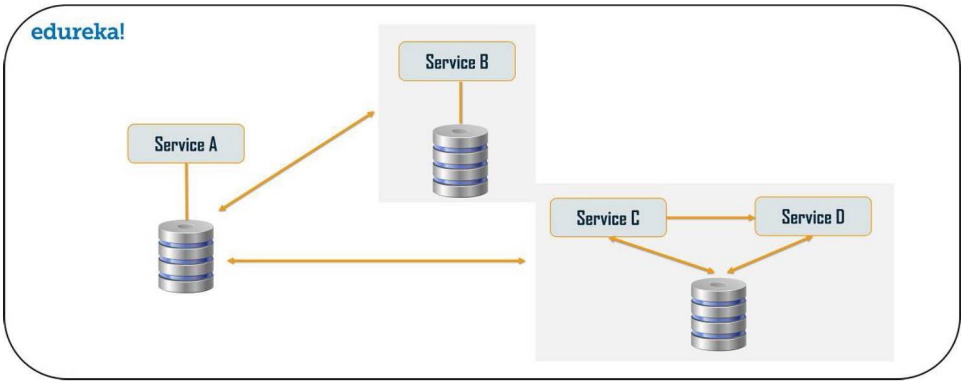

Рисунок 1.7 – типова схема паттерну Shared Data, на якій представлені сервіси B, C та D використовують єдине сховище даних, що надається сервісом A. Додатково, окреме сховище даних групи сервісів C та D є також об'єднаним

6. Event Sourcing: передбачає застосування моделі подій для взаємодії між сервісами та зовнішнім середовищем. Через події різного типу відображаються зміни до стану як індивідуальних сервісів, так і стану системи у цілому. Цей паттерн використовується як транзакційний, забезпечуючий більш високу цілісність системи та її даних у аварійних ситуаціях, дозволяючи відновити стан по журналу подій;

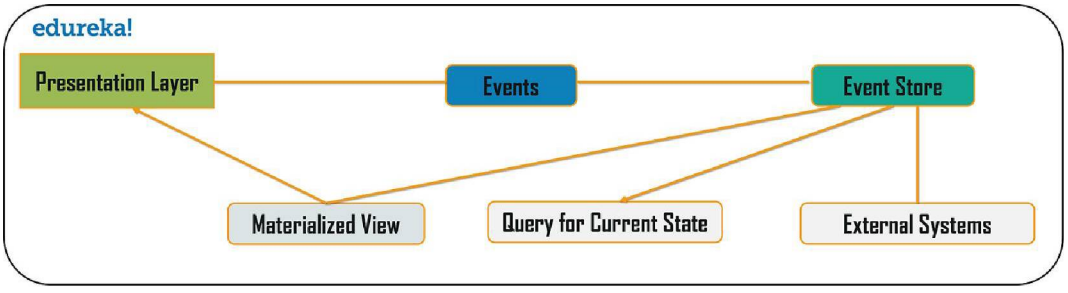

Рисунок 1.8 – типова схема паттерну Event Sourcing

У комплексних клієнт-сервісних рішеннях частим є присутніть більш ніж одного паттерну, включаючи їх об'єднання. Особливо це присутнє для програмних рішень, що надають послуги з систематизації та управління проектами, що включають группи поєднаних сервісів, за типом SaaS.

1.2.5 Використання у рішеннях типу SaaS.

Застосування мікросервісних рішень є дуже популярним для рішень типу SaaS, у яких користувачам надається можливість замовити налаштований сервіс, наприклад СУБД або розміщення для веб-сайтів. Визначенням "сервіс як сервіс" мається на увазі саме послуга та інструмент, яким надається конфігурація та розміщення сервісу, або платформи для використання сервісу користувачем. Основним методом забезпечення стабільної роботи таких послуг є обов'язкова плата у вигляді календарної підписки, або плати за використан ресурси.

Для забезпечення роботи таких сервісів-послуг виникає необхідність у рішенні, що складається з декількох основних компонентів, забезпечуючих наступну функціональність:

- Управління обліковими записами, статусом підписки та інформацією щодо обраного тарифного плану;
- Управління процесом оплати та обробки персональних даних (може бути виконано за допомогою зовнішнього сервісу);
- Конфігурація та контроль за роботою цільового сервісу, у тому числі його розміщення;
- Надання користувачу контрольованих засобів доступу до сервісу, та/або можливість завантажувати дані для обробки та проведення довільних операцій.

Відповідно до кожної частини функціональності можна віднести індивідуальний сервіс, що проводить операції з локальними, ізольованими даними, а для зовнішньої взаємодії використовує функції API, що є

стандартизоваими між усіма сервісами системи. Ізоляція функціональності та використання паттернів дизайну системи надає переваги стійкості до помилок та підтримки довільної кількості та типів клієнтів, що є важливими для розміщення сервісу та надання послуг найбільш різноманітними способами зі збереженнями максимальної доступності та спільності незалежно від представлення.

Найважливішим моментом для прийняття до уваги є контроль методів використання ресурсів та сервісів, що надаються у рамках послуги, виключення методів виходу за обмеження або зловживання можливостями для обходу обмежень тарифного плану або інших, встановлених постачальником послуги.

1.3 Методології та засоби командної розробки мікросервісних рішень.

Розробка та випуск косплексних програмних рішень на основі архітектури мікро-сервісів потребує багатьох складових, включаючи апаратні та програмні ресурси, службові сервіси та інструменти, що входять у систему програмного забезпечення для ведення циклу проектування, розробки та підтримки цільових продуктів. Систематизація та стандартизація процесу розробки програмних продуктів є одним з головних методів оптимізації витрат часу, ресурсів, та автоматизації рутинних операцій на багатьох етапах життевого циклу продукту.

1.3.1 Основні принципи інтегрованої роботи.

Процес розробки, проектування та підтримки програмного продукту поділяється на наступні категорії:

- Управління метаданими проекту: назва, опис, включені компоненти (сервіси, клієнти), конфігурація інструментальних сервісів;

- Управління планом роботи: розподілення завдань по виконавцям, статусу виконання, та групування завдань по окремим категоріям (функціональність, компонент, чи типове застосування), встановлення графіків та планів завершення завдань, досягнення контрольних точок у загальному процесі розробки;
- Управління програмним кодом: централізоване сховище коду, підтримка історії змін та дерева версій для забезпечення роботи над окремими частинами системи продукту;
- Автоматизація процесів тестування, збірки та розгортання продукту: надання інструментів для налаштування та контролю за процесами, як централізованими, так і індивідуально для кожного розробника на робочому місці для тестування та відладки;
- Збереження та обробка даних телеметрії, проведення аналітичних робіт та будування трендів щодо подальших дій, включаючи усунення можливих технічних проблем або додання нових функцій до ПЗ;
- Ведення докуметації щодо робочого процесу, як спільного для декількох продуктів (включаючи технічну документацію), так і унікальну для кожного;
- Доступ до ресурсів, включаючи інструменти, сервіси та документацію з мінімумом первинних налаштувань;
- Ідентифікація та захищений доступ до розділених даних, для уникання можливостей виотку інформації, що є особливо важливим для великих організацій.

Більшість цих елементів притаманна незалежно від організації, але є необхідною для ведення повноцінного процесу для складних клієнт-сервісних рішень через структуру ПЗ та кількість індивідуальних компонентів та їх формування.

1.3.2 Вимоги до інструментів розробки для забезпечення інтегрованої роботи.

Для забезпечення вимог описаних у п. 1.3.1, та додаткових, що можуть виникнути в організації, або у процесі роботи через встановлені вимоги до індивідуальних продуктів, система інструментарію повинна мати наступні якості:

- Доступність – інструменти повинні бути доступні на яомога більшому наборі платформ, включаючи мобільні, стаціонарні ОС, та веб-браузери. Інтерфейс для взаємодії має бути зрозумілим навіть для користувачів, що не мають розуміння у внутрішніх алгоритмах, що відносяться до визначеного елементу функціональності;
- Захищеність – протилежно доступності, доступ до приватних інструментів, використання яких може вплинути на роботу інших, або навіть на інфраструктуру продуктів або їх стан має бути захищеним з використанням методів авторизації та аутентифікації, подібним концепту 2FA, а також з додатковими обмеженнями, якщо взаємодія відбувається лише у локальній мережі;
- Розширюваність – інструменти повинні підтримувати можливість швидкого впровадження нових функцій, без втручання у їх наявну функціональність та необхідність перепроектування, що зробить складним застосування нової версії для роботи над існуючими продуктами та циклів розробки;
- Адаптивність – інструменти та сервіси повинні бути універсальними, базуватися на обраних стандартах (зовнішніх, або внутрішніх у організації), та диамічно змінювати доступний фукціонал у залежості від обраного продукту та виористовуваних технологій;

- Самодостатність – система інструментарію повинна мати як взаємодію між різними її частинами для забезпечення стандартизованої та безперервної можливості роботи над різними проектами та проектними базами, а й незалежність (або мінімальну залежність) від зовнішніх факторів, дозволяючи використання всіх (або більшості) наявних можливостей навіть за умови роботи на індивідуальній машині або у локальній мережі організації, так званій “air-gapped network”.

1.3.3 Автоматизація та стандартизація процесів розробки.

Розвиток можливостей та необхідності автоматизації різних елементів процесу розробки ПЗ є важливою частиною спрощення процесів та методів будування комплексних програмних продуктів. Саме через систематизацію індивідуальних інструментів та створення об’єднаних внутрішніх сервісів, що є еволюцією ПЗ, прив’язаного до індивідуального проекту або технологій, на якій він базується, до узагальнення функціональності та розширення для застосування можливостей без необхідності адаптації або створення інструментів з нуля. До найпоширеніших прикладів таких сервісів та інструментів можна навести наступні:

- Панелі статусу, статистики, відображення загальної інформації щодо проекту або його частин для швидкого розуміння статусу роботи над проектом та можливих подальших дій;

Рисунок 1.9 – приклад інструменту з панелями інформації для типового проекту, що відображає інформацію щодо завданнями розробки, статистики програмного коду, дані збірки та тестування та статусу середовищ розгортання компонентів проекту

- Автоматизований прийом, обробка та збереження інформації щодо програмних помилок для випущених продуктів. Такі інструменти надають можливість швидкого аналізу та діагностики проблем та причин їх виникнення, відповідно до версії та типу програми, що входить до продукту. Отримана інформація може бути корисна для особливих випадків, коли помилка виникає лише за унікальних умов, наприклад на одному типі пристрою або за відсутності інтернет-з'єднання. Використання ручного способу отримання інформації від користувачів було би більш затратним по витраченому часу з обидвох сторін (включаючи час на ручну обробку запитів та відкидання недостовірних чи невідповідних до продукту), а

також мало би меншу кількість ідентифікаційної інформації щодо платформи та середовища виконання кінцевого продукту;

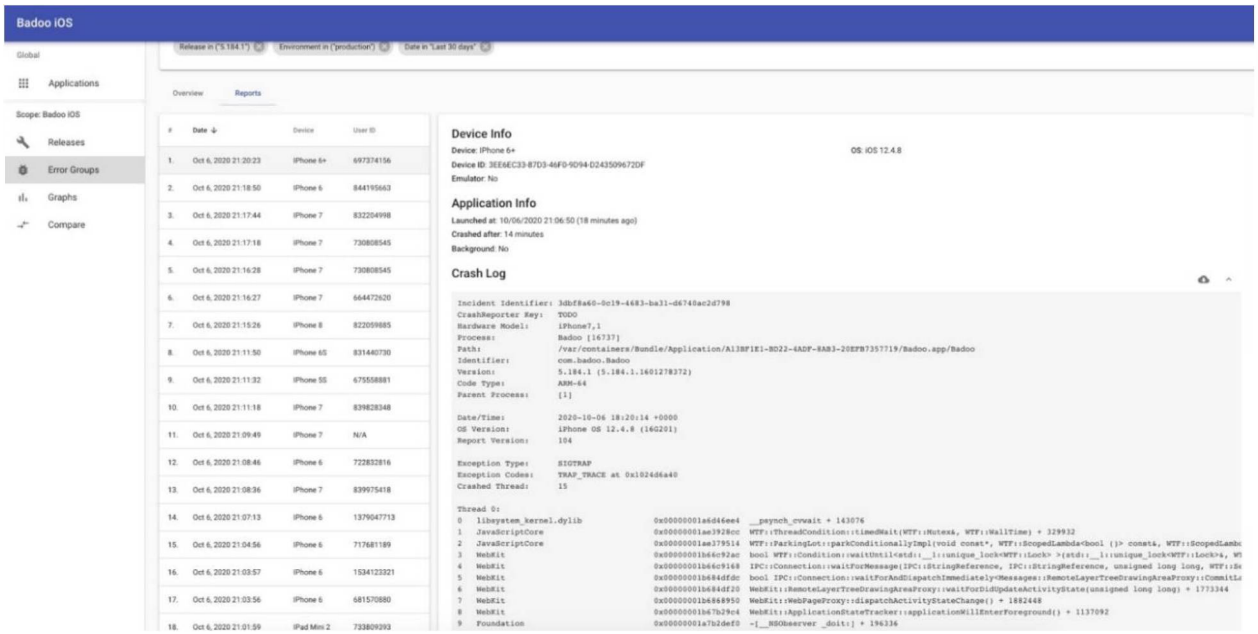

Рисунок 1.10 – приклад інструменту з відображенням інформації щодо зареєстрованої програмної помилки, отриманої та обробленої автоматичним шляхом від спеціально розробленого компоненту у кінцевому продукті

- Автоматизація та централізація управління частинами інфраструктури, що забезпечує роботу з зміни конфігурації або розгортання нових елементів (включаючи СУБД, віртуальні машини або ізольовані контейнери, підключення сховищ) з єдиного ресурсу та використовуючи спільні механізми незалежно від цільової платформи, типу ресурсу, або програмного продукту. Інструменти даної категорії також можуть надавати можливість програмувати роботу з елементами інфраструктури, спираючись на інформацію від інших сервісів, або на зовнішнє середовище (включаючи фактори, що можуть впливати на подальший стан компонентів та сервісів, що входять у склад продукту).

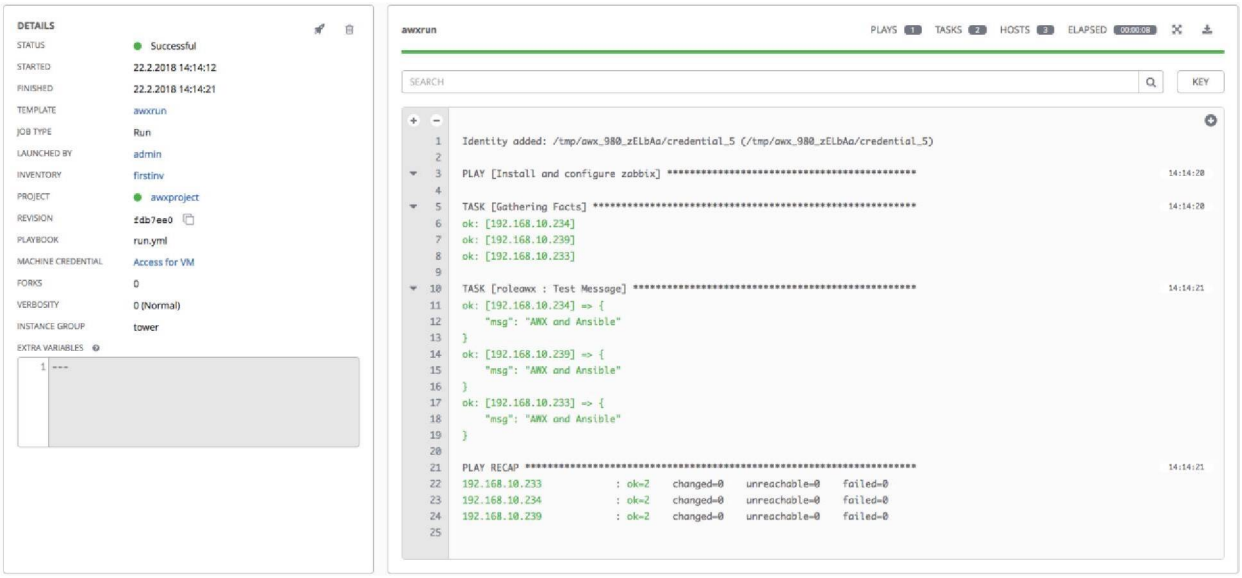

Рисунок 1.11 – приклад інструменту з автоматизованим виконанням списку операцій на 3 машинах, з параметрами задачі (групи операцій) зліва, та виводом результатів у консолі с правої сторони

1.3.4 Методологія DevOps.

DevOps – методологія ітеративної циклічної розробки програмних продуктів, що базується на щільній інтеграції програмних інструментів та сервісів у процес розробки. Застосування таких засобів є притаманним на кожному етапі циклу, серед яких є 7 основних [9]:

1. Розробка програмного коду, аналіз характеристик для подальшої його оптимізації;
2. Використання інструментів CI/CD для проведення автоматизоваих збірок варіантів продукту для тестування або пакування для подальшого розповсюдження та/або розгортання;
3. Використання інструментів автоматизованого тестування, що виявляють можливі помилки у коді та ризики різного рівня, включаючи безпекові та бізнес-логіки;

4. Пакування програмного коду та ресурсів, відправлення отриманих архівів до централізованого сховища/репозиторію артефактів;
5. Випуск продукту до споживання, включно з затвердженням, використовуючи стандартні, спальні інструменти;
6. Керування інфраструктурою, на якій працюють компоненти програмного продукту, у тому числі зміна конфігурації елементів, та представлення об’єктів інфраструктури як код;
7. Застосування інструментів та сервісів для обробки телеметрії та прийняття відгуків від користувачів для формування трендів щодо роботи продуктів.

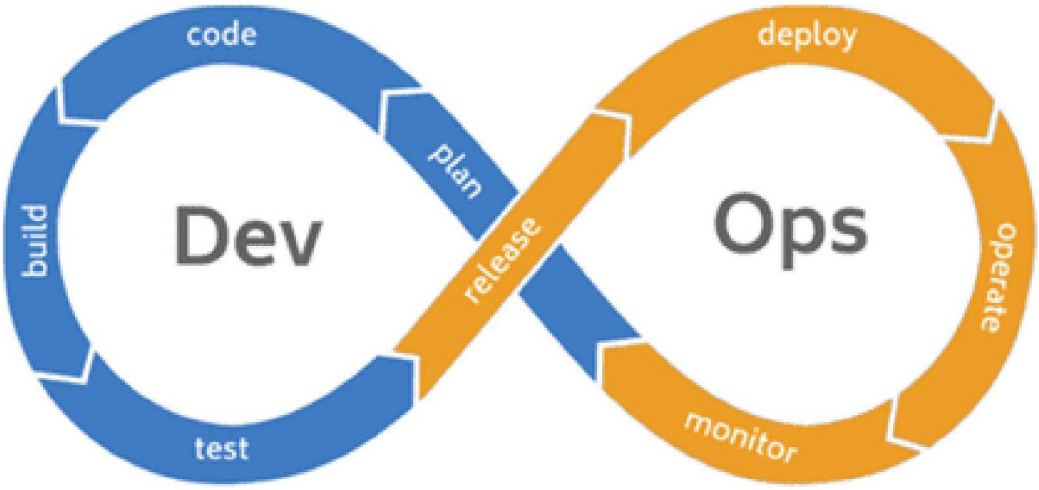

Рисунок 1.12 – концептуальна модель циклу розробки з застосуванням принципів DevOps

Основними цілями, що є одними з головних причин впровадження DevOps у процесах розробки ПЗ є наступні:

- Витрата меншої кількості часу та інших ресурсів порівняно з альтернативними методологіями для випуску нових продуктів;

- Покращення якості випускаємого ПЗ через використання інструментів для аналізу та швидкого виправлення проблем, та тестування таких виправлень;
- Мінімізація часу простою та обслуговування сервісних частин продукту за рахунок автоматизації процесів та більш швидкого реагування на виникаючі помилки ПЗ;
- Більш швидка ітерація над розробкою та впровадженням нових функцій та розширень у ПЗ існуючого у користуванні програмного продукту.

Впровадження механізмів DevOps та відповідних інструментів є ключем для оптимізації задіяних ресурсів та кількості відповідальних осіб, а також точок сбоїв, що можуть вплинути на результат наступних за ходом етапів роботи над проектом. Через виконання однакових за алгоритмом процесів, результат роботи буде завжди (або у більшості ситуацій) детермінованим.

Висновки до розділу 1.

Необхідність у систематизації процесів розробки програмного забезпечення та інновації, які торкаються як прикладних (розробка нових сервісів та інструментів), так і теоретичних частин є результатом розвитку технологічного прогресу, та підвищення складності та комплексності цифрових послуг, а також нових обмежень та вимог, що представляються перед подібними проектами. Перехід від монолітних архітектур до розподілених не є швидким та обов'язковим, але є поступовим, що залежить від побудови продукту та його вимог.

РОЗДІЛ 2.
ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ, ВИЗНАЧЕННЯ ЗАДАЧ РОБОТИ

Для розуміння області, у якій буде зосереджена практична частина роботи, є небхідність встановлення ситуації навколо технологічних рішень теперішнього часу, в особливості причини вибору того чи іншого підходу, а також їх переваги та недоліки.

Проектування та розробка інноваційного рішення має у собі прийняття уваги важливих деталей у наявних рішеннях, що потребують виправлення або перетворення для більш покращеної інтеграції у рамках однієї системи. Формування порівняльної картини щодо аналогів на ринку, та виявлення основних причин застосування саме моделі self-hosted, та наведення прикладів недоліків у інструментах та сервісах розробки та проектування допоможе сформувати загальну картину дослідження, та відповісти на наступні головні питання:

1. Чи є актуальним та популярним застосування ПЗ, розроблених для розміщення на власній інфраструктурі?
2. Які причини такого вибору?
3. Чи є вибір self-hosted рішень більш економним та вигідним порівняно з використанням публічних SaaS рішень та інших спільних продуктів?
4. Які основні недоліки або особливості мають наявні рішення обох категорій, усунення або зміна принципу роботи яких може бути корисною для використання подібних інструментів у рамках об’єднаного рішення з додатковими перевагами?
5. Чи є сенс у розробці нового рішення, використовуючи наявні принципи та функціональність, що представлена у існуючих рішеннях, застосовуючи лише їх об’єдання та інтеграцію, або

проектування унікальної моделі загалом (чи окремих компонентів) може бути більш корисним для досягнення поставленої мети?

2.1 Дослідження актуальності проекту.

Перехід до більш щільного та повсюдного впровадження інструментарію та інших засобів програмної автоматизації та управління процесами зумовлений багатьма причинами, але з основних можна виділити наступні [10, 11]:

- Підвищення різноманітності послуг у цифровому середовищі, у зв'язку з впровадженям технологій та доступу до мережі у різних сферах життя;
- Підвищена необхідність у моніторингу та обслуговуванні сервісних ресурсів продуктів у зв'язку з пандемією та високою навантаженістю на мережі через використання продуктів віддаленої роботи;
- Розвиток можливостей категорій продуктів, особливо соціальних мереж, створення так званих "хабів" – програмних рішень що виступають хостом для сторонніх сервісів;
- Глобалізація цифрових продуктів, необхідність підтримки їх роботи та адаптації для багатьох країн світу (включаючи локалізацію ресурсів та функціональності);
- Розвиток апаратних можливостей платформ, відкриття нових категорій для продуктів, необхідність у найманні спеціалізованих розробників та створенні або застосуванні інструментів, що адаптовані під роботу з цільовими платформами.

Дедалі більше організацій займаються розробкою більш ніж одного продукту одночасно, або продуктів, що включають комплексну архітектуру з великою кількістю компонентів, тому для них постає питання використання

стандартизованих сервісів та інструментів для організації робочого процесу у цілому та забезпечення централізованого контролю та автоматизації, без подальшої потреби у виконанні ручних операцій та ризиках випадкового пошкодження внутрішніх ресурсів або навіть кінцевих продуктів.

2.1.1 Постановка вибору – основні питання та критерії.

Формування вибору однієї чи іншої категорії рішень при проектуванні та розробці нового інструментального ПЗ, що відноситься до єдиної системи залежить від наступних критеріїв:

- Фінансові витрати – включаючи плату за супутне програмне та апаратне забезпечення, тарифні плани підписочних сервісів;
- Обсяг доступних функцій та категорій сервісів й інструментів;
- Область застосування, спеціалізація (якщо визначена для продукту);
- Можливості інтеграції з іншими рішеннями, як вбудовані, так і за допомогою моделей розширень (включаючи API сервіси та бібліотеки програмного коду);
- Наявність документації, її обширність та повнота висвітлення окремих частин (наприклад, опис інтерфейсу API, або інструкції з використання функції);
- Можливі обмеження при використанні.

2.1.2 Причини та цілі використання хмарних інструментів та сервісів корпорацій, основні складові витрат.

“Хмарні” інструменти та сервіси слугують як продукти, розроблені для використання великою кількістю користувачів, без прив’язки до обмежень або правил одної організації, яка їх використовує. Одною з основних відмінностей є те, що такі обмеження встановлюється компанією, що надає доступ до самого інструменту, або посгули з його використання. Такі обмеження можуть бути

зумовлені бізнес-політикою, станом індустрії, або іншими причинами, та накладаються на всіх користувачів. Деякі обмеження або зменшуються у їх впливі, або зовсім знімаються, якщо використовується більш дорогі тарифні плани, які зазвичай призначені для Enterprise застосування (рівня підприємства зі специфічними потребами та фінансовим бюджетом).

Серед можливих обмежень та їх причин можна виділити:

- Дозволені типи контенту – через юридичні зобов'язання та виконання законів країни, де розміщена компанія, що надає послуги;
- Максимальне використання сховища, ОЗП або процесорної потужності – через необхідність сплати за використання кожного ресурсу та запобігання перебільшення витрат на них;
- Неможливість прямого доступу до ОС, на якій розміщується сервіс/інструмент – через ризики встановлення вірусів або необмеженого використання ресурсів постачальника сервісу для інших цілей, у тому числі зловмисних;
- Неможливість довільного вибору та контролю версій сервісу/інструменту – подібні обмеження, що представляють собою обмеження динамічності налаштувань, можуть бути пов'язані як з безпековими причинами, так і з бізнес-пріоритетами, або необхідністю забезпечувати надійність роботи сервісу для якомога більшої кількості користувачів, за допомогою використання перевірених версій, що не завжди можуть бути останніми.

На прикладі GitLab можна побачити використання розділених тарифних планів з розширенням набору можливостей у більш дорожчих.

Три основні плани поділяються на "Безкоштовний" (Free), "Преміум" (Premium), та "Ультімейт" (Ultimate). Кожний план розрахован на

використання різними групами користувачів, відносно їх потреб та фінансового бюджету, наприклад план Ультімейт надає спеціалізовані можливості для організацій щодо додаткового контролю за безпековими ризиками, що є дуже важливим для цілісності даних [12].

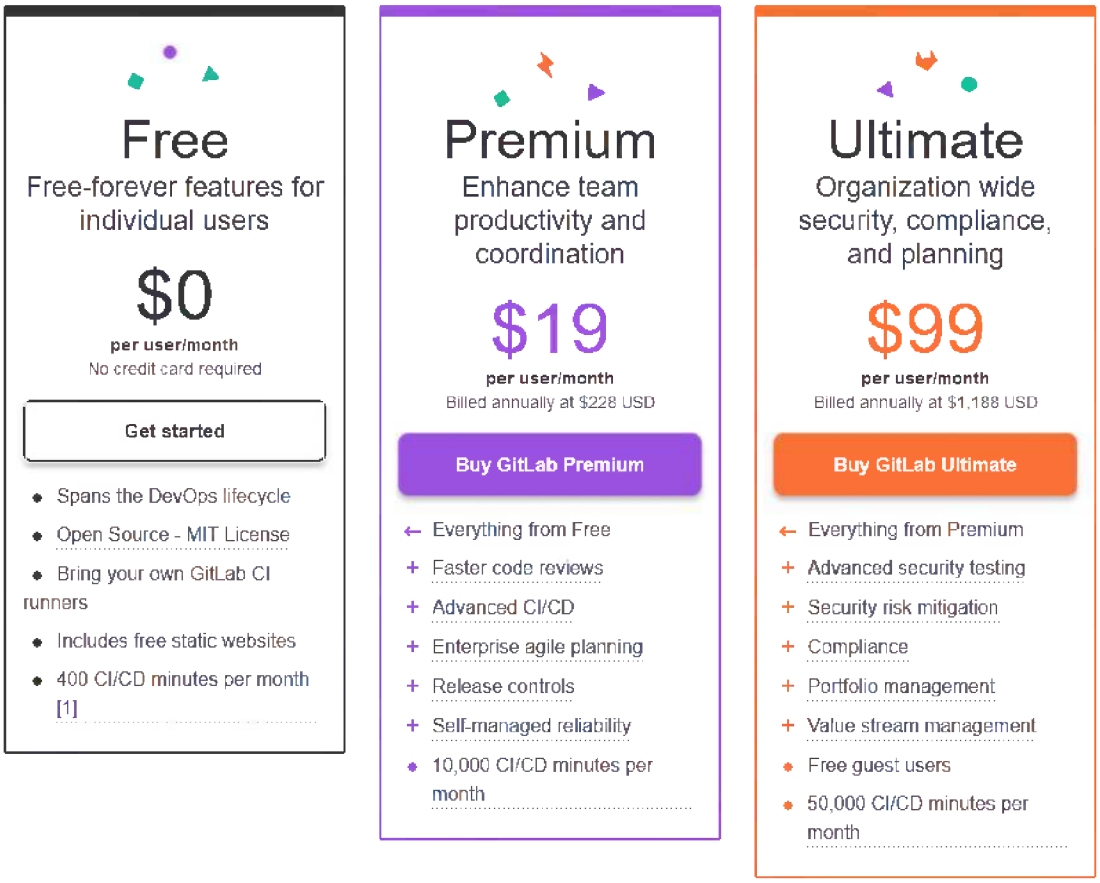

Рисунок 2.1 – приклад надання сервісу у якості публічної спільної послуги з розділенням функціональності на прикладі GitLab

Одною з голових причин вибору саме такої категорії інструментальних рішень, навіть для великих організацій стає зручність. Поняття зручності у даному контексті включає у себе наступні елементи:

- Принцип “plug-and-play”, тобто від реєстрації до початку роботи з продуктом необхідно мінімальна кількість часу та ручних налаштувань;
- Інтеграція з іншими популярними продуктами, такими як Slack або GitHub;

- Управління інфраструктурою зазвичай є відповідальністю не користувача, а організації, що надає сервіс;
- Користувачу не потрібно знати як працює сервіс зсередини через відсутність можливості та необхідності внесення глибоких змін;
- Отримання підтримки щодо використання та роботи сервісу від окремої команди, або на формуах з іншими користувачами, що пришвидшує вирішення популярних можливих проблем, порівняно з самостійною діагностикою та спробами виправлення.

2.1.3 Причини та цілі використання самостійно розміщених та розроблених інструментів та сервісів, основні складові витрат.

У свою чергу, принципово, сервіси та інструментне ПЗ спроектоване для розміщення на власній інфраструктурі, та порівняно до так званих “hosted” рішень, використовується для забезпечення наступних можливостей:

- Незалежність від зовнішніх сервісів, можливість застосування комплексу як самодостатнього рішення у закритому середовищі;
- Відкритий доступ, дозволяючий довільну конфігурацію та внесення змін до програмного забезпечення або зовнішньої середи, що пов’язана з продуктом;
- Більш детальний контроль за використанням ресурсів, виділяючи для роботи лише необхідні (як програмні, так і апаратні);
- Більше варіантів розгортання сервісів, у тому числі на платформі на основі контейнерів (наприклад Docker або Kubernetes), забезпечуючи стандартизовані можливості моніторингу та конфігурації інфраструктури;
- Прив’язка до та більш глибока інтеграція з внутрішніми сервісами, що можуть бути не підтримуваними у хмарних рішеннях (наприклад, аутентифікація за методом OIDC/OAuth 2.0).

У випадку використання рішень, що базуються на моделі self-hosted, фінансова складова полягає лише у витратах на апаратні ресурси та їх підтримку (побутові витрати), та на супутне програмне забезпечення, необхідне для функціонування розгортаємого сервісу або інструментарію (ліцензія для ОС та інші).

2.1.4 Порівняння переваг та недоліків підходів.

У порівнянні критеріїв двох основних моделей розповсюдження продуктів, та виділені особливостей, що обгрунтовують застосування тих чи інших рішень була сформована таблиця 2.1, що відображає загальні переваги та недоліки, притаманні більшості інструментальних та сервісних рішень тої чи іншої моделі.

Таблиця 2.1

Порівняльна таблиця критеріїв для типових хмарних та self-hosted продуктів

Критерій	**Хмарні (cloud-hosted) рішення**	**Самостійно розміщені (self-hosted) рішення**
Фінансові витрати	Підписочні тарифні плани з фіксованою ціною за період (місяць/рік), або плата за використання ресурсів, що надаються кожним сервісом продукту	У більшості випадків – безкоштовне використання, для деяких комерційних рішень є необхідність одноразова оплата ліцензії; додатково витрати за власні ресурси інфраструктури
Категорії та обсяг доступних сервісів	Залежить від спеціалізації (примітки нижче)	Залежить від спеціалізації, але з основних входять: управління обліковими записами, веб-додаток, внутрішня статистика, портал обслуговування системи для адміністраторів

Таблиця 2.1 (продовження)

Критерій	Хмарні (cloud-hosted) рішення	Самостійно розміщені (self-hosted) рішення
Спеціалізація	Набори сервісів та інструментів для окремих робочих процесів, або загального призначення	Набори сервісів та інструментів для окремих робочих процесів, або окреме інструментальне ПЗ
Інтеграції	Вбудовані інтеграції з іншими популярними хмарними сервісами, включаючи месенджери, збору статистики та телеметрії, з суміжними продуктами компанії-виробника	Здебільшого інтеграції з іншими self-hosted продуктами, а також підтримка веб-хуків (webhooks) для інтеграції з власними проектами через протокол HTTP або подібні
Обмеження	• відсутність стандартизованих методів управління та доступу до внітрішніх сервісів (окрім деяких випадків, наприклад віртуальні машини), • обмеження по контенту, • обов'язкове прийняття політики конфіденційності виробника, • обов'язкове підлючення до інтернету.	• політика "as-is" для безкоштовних продуктів (відсутність офіційної підтримки від розробника), • обмежена функціональність у порівнянні з комерційними хмарними продуктами (наприклад, відсутніть розширених функцій або деяких компонентів, що потребують комерційних ліцензій), • залежність від обмеженої кількості підтримуваних платформ та системних вимог, • необхідність самостійного обслуговування та оновлення ПЗ до останньої версії (у більшості випадків)

Таблиця 2.1 (продовження)

Критерій	Хмарні (cloud-hosted) рішення	Самостійно розміщені (self-hosted) рішення
Додаткові особливості	• Офіційна підтримка від виробника, • Вбудовані механізми управління оплатою за використання та ліцензіями в рамках облікового запису, • Задачі щодо оновлення ПЗ, проведення обслуговування та управління інфраструктурою полягаються на виробника	• Довільний доступ до програмного коду та можливість внесення змін для приватних цілей (у більшості випадків), • Повий контроль за роботою ПЗ та даними, • Незалежність від зовнішніх ресурсів (у більшості випадків).

Примітки:

- Категорії сервісів, що можуть надаватися у рамках одного хмарного продукту, або окремо, поділяються на наступні:
  - Управління обліковими записами користувачів – механізми та інтерфейси реєстрації/авторизації та керування доступом до інших сервісів залежно від налаштувань;
  - Обробка та збереження даних з використанням різних методів зберігання (БД, файлова система та інші);
  - Система контролю версій програмного коду та контенту – зазвичай базується на концепті Git та надає як веб-додаток, так і можливість підключення стандартного Git-клієнту;
  - Система доставки контенту (CDN) - розповсюдження даних з дублюванням до кінечних точок для зменшення часу завантаження та кількості запитів до головного серверу;

- Системи CI, що надають можливості по автоматизації процесів збірки, тестування та випуску програмних продуктів;
- Інструменти трекінгу часу, задач та програмних помилок;
- Віртуальні машини;
- СУБД;
- Централізований доступ до багатьох пристроїв, незалежно від кінцевої платформи;
- Управління та контроль стану внутрішніх віртуальних мереж;
- Застосування ізоляції виконання різних сервісів та їх копій з використанням “контейнерів”;
- Та інші.

- Наявні продукти за приципом застосування поділяються на інтегровані системи та сервісні пакети. Головні відмінності:
  - Інтегровані системи – спеціалізовані комплекси ПЗ, що затосовуються для виконання вузького набору завдань, та мають обмежений набір інструментів та сервісів, що інтегровані між собою без можливості внесення змін окремо до кожної частини (зазвичай);
  - Сервісні пакети – набори інструментів та сервісів, застосування яких є на розсуд користувача, але між собою вони можуть бути інтегровані та зазвичай надаються у рамках одного рішення (такі як MS Azure, Amazon Web Services та Google Cloud Platform). Призначення таких систем є загальним, а наявність більш широкого вибору забезпечує налаштування для відповідних вимог замовника/користувача.

2.1.5 Огляд та порівняння складових існуючих рішень.

Зважвючи на те, що комплексні системи для організації та ведення процесу розробки, включають різні типи інструментального ПЗ, які

співпрацюють між собою, то виникає сенс порівняння двох наявних рішень на компонентному рівні, відносно типових застосувань та необхідного ПЗ для процесу DevOps.

Для порівняння було обрано GitLab та Gitea. Серед головних спільних особливостей:

- Спеціалізація на управління та контроль за програмним кодом на базі коцепту Git, інші інструменти та сервіси є залежними;
- Обидва рішення підтримують розгортання у власному середовищі (self-hosted);
- Клієнтські інструменти мають спільний веб-інтерфейс, що забезпечує портативність та можливість використання незалежно від платформи (але з деякими обмеженнями через особливості веб-додатків);
- Обидва рішення є безкоштовними та без критичних обмежень. Можливі обмеження можуть бути зумовлені лише дизайном або архітектурою систем у цілому.

Для порівняння щодо загальних складових програмного забезпечення та його сумісності, були визначені наступні критерії:

- Підтримувані платформи та ОС;
- Підтримувані процесорні архітектури (офіційно);
- Процес оновлення;
- Підтримувані типи зовнішних баз даних;
- Вимоги до ресурсів системи;
- Наявність мобільних додатків.

Для порівняння щодо можливостей використання рішень у рамках процесів DevOps, були визначені наступні критерії:

- Підтримка CI/CD (автоматизованої конвеєрної збірки та розгортання, з підтримкою динамічних конфігурацій);

- Розділення проектів (за визначенням систем) на віртуальні організації або групи/команди;
- Копіювання репозиторіїв проектів з інших систем, дзеркала репоиторіїв;
- Шаблонізація для різних функцій;
- Система трекінгу проблем для проектів;
- Організаційні дошки для завдань (Kanban-стиль);
- Управління обліковими записами, система аутентифікації/авторизації;
- Wiki-документація для проектів;
- Статичний хостинг для сайтів проектів;
- Підтримка інтеграцій;
- Трекінг часу для завдань;
- Система управління артефактами;
- Локальні репозиторії контейнерів (сумісних з Docker та Kubernetes);
- Локальні репозиторії пакетів (npm, NuGet, та інших систем управління пакетами);
- Можливість зміни стилю та елементів інтерфейсу;
- API інтерфейс для зовнішньої взаємодії.

У таблиці 2.2 приведені результати порівнянь по кожному критерію, наведеному вище [13, 14].

Таблиця 2.2

Порівняльна таблиця критеріїв для систем ПЗ щодо організації процесів розробки програмних продуктів

Критерій	GitLab Community Edition	Gitea
Підтримка платформ та ОС	Linux; Docker, Kubernetes	Windows, Linux; Docker
Підтримка архітектур	x64	x86, x64, ARM64
Процес оновлення	Вручну, з підтримкою нульового простою [16]	Вручну, з перезапуском [15]
Підтримка зовнішніх БД	PostgreSQL, MySQL (до версії 12.1)	SQLite, MySQL, PostgreSQL
Системні вимоги	Мінімум 4 ГБ ОЗУ, 4-х ядерний процесор	Мініумум 512 МБ ОЗУ, 2-х ядерний процесор
Мобільні додатки	Неофіційні (iOS, Android)	Неофіційні (iOS, Android)
Підтримка CI/CD	+, включаючи управління конвеєрами проектів та агентами збірки	-
Віртуальні організації	+	+
Дзеркала репозиторіїв	+	+
Шаблонізація	+ (у трекінгу проблем)	+ (репозиторії проектів, трекінг проблем)
Трекінг проблем ПЗ	+	+
Організаційні дошки	+	+
Управління обліковими записами	+ (вбудований, OIDC/OAuth 2.0, SAML, LDAP)	+ (вбудований, OIDC/OAuth 2.0, SAML, LDAP)
Wiki-документація	+	+
Статичний хостинг	+ (пдтримка зовнішніх доменів)	-
Підтримка інтеграцій	+ (вбудовані та веб-хуки)	+ (вбудовані та веб-хуки)
Трекінг часу	+	-
Управління артефактами	+ (включено у інструменти CI/CD)	-

Таблиця 2.2 (продовження)

Критерій	GitLab Community Edition	Gitea
Репозиторії контейнерів	+	-
Репозиторії пакетів	+	-
Свій дизайн та інтерфейс	-	+
API взаємодії	+	+

Загалом, за результатами порівняння, можна побачити, що Gitea є більш вузькою по спеціалізації системою, але включає у себе можливість внесення змін до веб-інтерфейсу інструментів, тим самим більш зільно об'єднуючи дану систему з іншими внутрішніми сервісами та інструментами. У той же час, нестача голових компонентів (такі як автоматизація збірок та розгортань, хостинг статисних сайтів, та управління пакетами) не дозволяє обрати Gitea як систему, що задовольняї поставлені до проекту вимоги.

GitLab, у свою чергу, відповідає більшій кількості вимог та має позитивну оцінку по більшості критеріїв порівняння, але у свою чергу, баланс між функціональністю та сумісністю та легкістю системи відходить у більш негативний результат для останнього, через відсутність сумісності з ARM64 платформами, та високими вимогами до апаратного забезпечення. Такі фактори також заважають обрати GitLab для використання готовою системою.

2.1.6 Обгрунтування вибору методу self-hosted для розробляємого проекту.

Базування проекту на наявних хмарних рішеннях було би доцільним через можливу економію часу через відсутність необхідності на розробку з нуля, та застосування перевірених й популярних сервісів, що полеглшило би поріг входу розробникам-користувачам. Але у такому підході є декілька недоліків, що роблять його використання неможливим:

- Вартість – з обмеженим бюджетом постає необхідність вирішення питання будування складної системи з мінімальними витратами. Зважаючи на результати долідження та порівняння фінансового критерію, можна встановити, що очевидним є вибір самостійного рзоміщення через гнучкість та необмеженість у доступних варіантах (як у виборі апаратної, так і програмної платформи). Особливо складним є обмеження деяких функцій (наприклад, Single sign on) за більш дорожчими тарифами;
- Обмеження щодо доступної функціональності;
- Неможливість локального застосування або без активного з'єднання з інтернетом, складний процес контролю даних та їх подальшого використання;
- Відсутність гнучкості щодо налаштувань та внесення змін до алгоритмів роботи сервісів, розширення функціональності;
- Високі системні вимоги та обмежений тип підтримуваних платформ для клієнтських частин (інструментального ПЗ), наприклад відсутність підтримки систем на базі Linux або архітектури ARM64.

Зважаючи на кількість та вплив недоліків, вибір інуючих рішень на базі хмарних продуктів є недоцільним та не є сміжним з вимогами до проекту.

2.2 Дослідження доцільності проектування моделі нової системи.

Основним питанням, що постає при формуванні нового процесу розробки, особливо у рамках DevOps, є вибір між застосуванням існуючих рішень, або створеням нового. Такий вибір залежить від багатьох обставин та вимог, поставлених перед цільовим проектом, для якого буде застосовуватися процес, або організацією у цілому. Важливим при проектуванні є встановлення балансу між використанням існуючих технологій (а саме self-

hosted рішень), за умови прийняття їх обмежень, та розробки нових, що вбачають у себе адаптацію до конкретних умов та вимог для застосуванні при розробці з новим процесом.

2.2.1 Причини та цілі розробки інструментів та сервісів розробки програмних продуктів для внутрішнього застосування.

Звісно, що конкретні причини є відмінними для кожного випадку, та навіть у рамках однієї організації можуть бути відмінності серед різних категорій інструментів та сервісів, та на рівні проектів. Але загалом, основні можливі причини для створення власних програмних інструментів є наступні:

- Обсяг використання та економія бюджету – залежно від цільового сервісу та запланованих обсягів використання, має сенс розробити власний додаток або сервіс, що заощадить кошти бюджету, у той же самий час надасть необхідні функції для розробників без втрат часу або у структурі робочого процесу;
- Особливі функціональні потреби – у залежності від цільового проекту або внутрішніх процесів, може з'явитися необхідність у створенні спеціалізованих інструментів для роботи, наприклад для обробки особливих форматів даних або збірки компільованих артефактів ПЗ для випуску, та інших варіантів;
- Питання безпеки – для деяких організацій є дуже важливим збереження конфіденційності даних та/або таємниці робочого процесу, тому використання існуючих рішень, навіть само-розміщених, може привнести можливі ризики витоку даних або навіть несанкціонованого доступу. Одним з останніх випадків, що торкнувся багатьох державних організацій та корпорацій у США, включаючи компанії рівня Microsoft, був пов'язаний зі зламом серверів продукту SolarWinds та розповсюдження версії ПЗ зі

встановленими вірусами для віддаленого доступу у внутрішні мережі [17].

2.2.2 Обгрунтування вибору розробки індивідуального рішення.

Порівняно з наявними рішеннями, як для самостйного розміщення, так і хмарних, проект даної роботи має достатньо переваг та особливостей, що можуть зумовити створення і використання саме нового ПЗ:

- Фінансова незалежність - єдині фінансові витрати, пов’язані з даним проектом, відносяться в основному до плати за споживання ресурсів апаратним забезпеченням, та закупівлею самого апаратного забезпечення (сервери та комплектуючі). Враховуючи широкий асортимент пропозицій та можливість точного визначення необхідних потужностей для роботи, є очевидною можливість економії засобів, порівняно з використанням готових рішень, які можуть бути не тільки менш відкритими для глибокого налаштування, а й також залежати від регулярної плати за надані послуги з використання цих рішень;
- Функціональні та архітектурні можливості - порівняно з використанням існуючих продуктів, розробка свого рішення з використанням супутних крос-платформних інструментів може дозволити застосування ПЗ не лише на одній окремій архітектурі та операційній системі, а й на декількох одночасно, що дозволяє зв'язувати окремі сервіси та інструменти у єдиний комплекс незалежно від наявних апаратних чи програмних ресурсів. У додаток до цього, за рахунок прямого доступу до програмного коду також можливо проводити розширення функціоналу для своїх потреб, тобто виробник є одночасно й користувачем;
- Більш гнучке масштабування в залежності від необхідності - за рахунок більш широкого доступу до конфігурації системи та

програмного коду, розробник та/або адміністратор має можливість налаштувати та адаптувати сервіси до роботи з різними рівнями навантаження, а також взаємодії з іншими сервісами та компонентами системи. Є можливість для створення будь-яких механізмів взаємодії з зовнішнім середовищем, що не є доступним для всіх продуктів-сервісів у корпорацій, або надається за додаткову плату.

Висновки до розділу 2.

У розділі 2 було проведено огляд основних категорій та методів надання послуг сервісів та інструментів, проведено порівняльні дослідження за спільними критеріями загальних методів (хмарних та само-розміщених продуктів), а також окреме порівняння двох існуючих комплексних систем для забезпечення ведення процесу розробки по методології DevOps. Було проведене дослідження щодо питання вибору між використанням існуючих хрваних рішень, або створення нового, а також дослідження щодо причин вибору само-розміщених та розробки індивідуальних внутрішніх інструментів окремими організаціями.

За разультатами досліджень та порівнянь, було встановлено та обгрунтовано причини розробки нового рішення на основі само-розміщеного комплексу інструментального та сервісного ПЗ, спираючись на економію бюджету, більш відкритий доступ, розширені можливості налатшувань та внесення змін, а також забезпечення більшого рівня безпеки та контролю за інформацією у системі.

РОЗДІЛ 3.
ОГЛЯД ВИМОГ, АРХІТЕКТУРНОЇ МОДЕЛІ, СТРУКТУРНИХ ЧАСТИН ТА МЕТОДІВ ПРОЕКТУВАННЯ КОМПОНЕНТІВ КОМПЛЕКСУ ПЗ

У експериментальній частині даної наукової роботи представлено проект моделі комплексу, що складається з набору інструментального та сервісного ПЗ, інтегрованого між собою, та заснованого на спільних архітектурних принципах та базових елементах. У процесі проектування були визначені вимоги до системи, включаючи технічні та функціональні, а також було обрано за результатами досліджень базові фреймворки та бібліотеки.

3.1 Постановка завдань для проектування та розробки моделей основних категорій ПЗ для внутрішнього застосування.

Визначення вимог до технічного рішення є одним з найосновніших кроків у процесі розробки. Для проекту були сформовані вимоги у цілому, а також окремо до окремих частин, включаючи окремі компоненти, їх призначення та необхідний функціонал, системні вимоги, а також основну структуру, включаючи спільні та унікальні логічні компоненти.

3.1.1 Визначення основних вимог для комплексу ПЗ та функціональності для цільових користувачів.

Розробляємий проект повинен представляти собою зв'язний комплекс програмного забезпечення, основною метою якого є поліпшення умов проектування, розробки, тестування та обслуговування інших програмних продуктів. Основні вимоги до системи у цілому:

- Універсальність - забезпечення можливості розробки та підтримки кінцевого продукту незалежно від впроваджених змін у його роботу, дизайн, або належність;

- Уніфікація - всі інструменти, документація та інтерактивні можливості взаємодії з ними повинні бути доступними на якомога великій кількості платформ та дозволяти користувачам системи отримувати доступ через єдиний інтерфейс;
- Собівартість - розробка даної системи повинна бути якомога дешевшою та використовувати, по можливості, безкоштовні компоненти, або ті, що мають відкритий програмний код;
- Інтеграція та розподіленість - система повинна дозволяти за потреби змінювати інструменти та деякий функціонал, який не потрібен для того чи іншого кінцевого програмного продукту. Видалення, або відключення, чи додавання інструментів до системи повинно мати стандартизований вид, щоб була можливість у майбутньому адаптувати до роботи з нею новий функціонал. У свою чергу, інструменти, що не є критичними для її роботи, повинні бути незалежно розміщені (включаючи використовувані ресурси, дані та програмний код) та мати мінімум залежностей з критичною частиною в плані взаємодії для недопущення проблем з усією системою коли один компонент виходить з ладу;
- Шаблонізація - система повинна підтримувати принцип “шаблонів” для різних інструментів та засобів проектування, розробки та обслуговування, що дозволить економити час у процесі підготовки різних частин кінцевого продукту до розробки або застосування. Такими шаблонами можуть бути шаблони сервісів, репозитаріїв програмного коду, документів, проектів з програмним кодом, проектів кінцевих продуктів загалом, тощо;
- Легкість у обслуговуванні - система повинна дозволяти її адміністраторам можливість зміни налаштувань без необхідності перезавантаження всієї системи або її компонентів, якщо ті зміни не є критичними та впливають на роботу платформи (апаратної чи

програмної), де розташована ця система. Компоненти системи повинні мати функціонал щодо забезпечення оновлення конфігурації автоматично та у фоновому режимі після прийняття змін від користувача, а оновлення системи чи її компонентів до нової версії повинні мати мінімальний вплив на її доступність та забирати мінімум часу на проведення самого процесу оновлення;

Для визначення функціональних вимог до проекту потрібно встановити різницю між призначенням кожної з категорії функцій:

- Для кінцевого продукту – функції, що виконуються сервісами та іншим програмним забезпеченням, що входить до кіцевих програмних продуктів, або застосовуються ними (якщо це є спільні ресурси або сервіси між декількома продуктами). Така функціональність визначається вимогами до продукту та представляє ідивідуально спроектовану бізнес-логіку, виключаючи спільні частини. Сервіси, що побудовані та призначені для виконання таких функцій працюють на базі ресурсів та сервісів, сформованих спеціально як основи (подібних до типу SaaS), що відносяться до наступної категорії;
- Для внутрішнього застосування під час проектування, розробки та підтримки продукту – призначені для застосування розробниками та іншими членами команд, що працюють над проектуванням, розробкою та підтримкою продуктів. До цих функцій відносяться відповідні основи DevOps, а призначення інструментів та сервісів, сформованих на їх основі, є тільки для внутрішнього використання та напряму не повинні бути доступні користувачам у рамках кінцевих випущених продуктів.

Цільова функціональність внутрішнього застосування є сміжною з типовими застосуваннями інструментарію у методології DevOps. Для даного

комплексу основні вимоги поділяються на наступні можливості, згруповані за спільною метою:

- Централізоване управління обліковими записами, захищеним доступом до даних та ресурсів:
  - створення/видалення/зміна інформації аккаунтів,
  - авторизація та аутентифікація користувачів для різних цільових інструментів та сервісів, що зареєстровані у БД як клієнти та підтримують даний мехінізм авторизації/аутентифікації,
  - використання різних типів облікових даих для підтвердження прав на доступ до ресурсу: електронна пошта, пароль, відбиток пальця, NFC-картка, додатковий секретний код та інші,
  - встановлення політик авторизації/аутентифікації у залежності від цільових ресурсів або виконуваних дій, включаючи:
    - мінімальну кількість символів паролю,
    - необхідність використовувати 2FA,
    - блокування через велику кількість неправильних спроб,
    - відключення/включення авторизації/аутентифікації для окремих ресурсів, дій або облікових записів.
  - наявність стандартизованого веб-інтерфейсу для двох частин:
    - адміністраторська, для виконання завдань управління та контролю,
    - клієнтська, для проведення операцій реєстрації/авторизації та аутентифікації до цільових ресурсів. Деякі елементи повинні адаптуватися до параметрів сессії, включаючи відображення

інформації про цільовий ресурс, заявлені ним дозволи, та адаптація дизайну веб-додатку до розміру єкрану пристрою та теми кольору,

- динамічна система дозволів для клієнтів, компартменталізація дій та даних, що відносяться до облікових записів,
- система аудиту, що повинна реєструвати як історію дій з обліковими записами для власників таких аккаунтів, а й окремо повідомляти як адміністраторам та і користувачам (у залежності від випадку) про підозрілі дії щодо аккаунту,
- використання похідного від OIDC/OAuth 2.0 механізму авторизації, з застосуванням стандартних параметрів, та з доданням власних значень параметрів, а також додаткових параметрів як до сессій авторизацій/аутентифікації, так і до "інформаційного документу" сервісу [18],
- наявність спільних клієнтських бібліотек, що взаємодіють з API сервісу та дозволяють цільовим клієнтам інтеграцію без необхідності впровадження взаємодії з нуля для кожного нового випадку.

- Централізоване управління метаданими та складовими проектів, що є основами для вироблених кінцевих програмних продуктів:
  - Віртуальні команди/організації (для забезпечення розділення робочого процесу між командами та/або продуктами):
    - Створення/видалення/модифікація інформації про групи, у тому числі склад учасників, права доступу до ресурсів, та прив'язані проекти
  - Робота над метаданими проектів:

        - створення/видалення/модифікація проектів, включаючи прив'язку до груп, логотипи, інші графічні дані, назву, кодове ім'я, тип, та інше,
        - автоматична генерація нових проектів та інформації щодо них, використовуючи шаблони.
    - Робота над проектною документацією:
        - генерація за допомогою шаблонів,
        - підтримка спільного редагування,
        - підтримка markdown-стилю,
        - експорт сторінок у PDF,
    - Управління складовими проектів:
        - включення/виключення наявних сервісів (з інших проектів або ресурсів),
        - включення/виключення інших типів ПЗ ("модулі" та "функції"),
    - Інтеграція з клієнтськими інструментами (включаючи інструменти від інших виробників, якщо можливо через системи плагінів) за допомогою API сервісу,
    - Забезпечення функціональності сервісу через веб-додаток та додаток для ПК, що дозволяє керувати параметрами робочої середи, додавати інші інструменти та програмні засоби для розробки та проводити операції над проектами без відкриття інших ресурсів.
- Сервіс управління та контролю за процесами та операціями програмних продуктів:
    - Є насамперед комплексною системою, включаюча веб-додаток, декілька сервісних компонентів та супутне ПЗ,
    - Управління розгортанням та випуском продуктів та супутних сервісів,

- Налаштування внітрушньої взаємодії між компонентами продуктів (у тому числі віртуальні мережі),
- Отриманя повідомлень про зміну статусу роботи компонентів продуктів, у тому числі з інтеграціями до інших інструментів або систем,
- Здійснення пошуку по ресурсам різних типів,
- Використання груп ресурсів для роздільного управління, у тому числі забезпечення захищеного доступу до окремих ресурсів, пов'язаних з проектами,
- Забезпечення прямого доступу до системи, на якій працює сервіс, через віртуальну консоль,
- Моніторинг параметрів роботи компонентів проектів, у тому числі ресурсів, будування статистики з можливістю експорту до інших систем та інструментів,
- Версіонування розгортань з збереженням історії та можливість відкату на довільну версію,
- Огляд інформації про запущені одиниці кожного типу сервісу, окремий моніторинг та конфігурація для кожного, включаючи збір лог-даних,
- Підтримка Service Discovery для динамічного оновлення стану сервісів або їх одиниць у системі,
- Застосування розширень зі сторони сервісів для надання додаткового функціоналу операторам у веб-додатку сервісу.

3.1.2 Визначення мінімально необхідних системних вимог.

Програмне забезпечення комплексу повинно задовольняти, згідно вимог, якомога більшу кількість платформ (програмних та апаратних). Це включає у себе наступну сумісність:

- Для інструментальної частини (клієнтські додатки):

- З більшістю популярних операційних систем: Windows, macOS, Linux (ПК), iOS, Android (мобільні), включаючи останні версії цих систем;
- З основними процесорними архітектурами: x64 (ПК), ARM64 (мобільні пристрої);
- Для спільних функцій: застосування або спільних крос-платформних реалізацій, або подібних на різних платформах, якщо немає портативної імплементації чи бібліотеки;
- Підтримка роботи на інтегрованих GPU з застосуванням мінімальної кількості оперативної пам'яті (відноситься до додатків для ПК, залежить від реалізації інтерфейсу користувача);
- Макимальне використання загальної оперативної пам'яті повинно бути не більше 512 МБ (окрім випадків проведення операцій з великою кількістю даних, що завантажені у пам'ять, або відображення комплексних інтерфейсів, таких як 3D-простор).

- Для серверних компонентів (сервіси та допоміжне ПЗ):
  - ОС: Windows, Linux;
  - Архітектури процесорів: x64, ARM64;
  - Підтримка роботи на платформах оркестрації контейнерів (віртуальних системах), або віртуальних машинах;
  - Мінімальні системні вимоги також повинні виконуватися залежними компонентами, такі як СУБД або окремим програмним забезпеченням;
  - У залежності від мети сервісу та кількості оброблюємих даних та/або запитів, використовувана оперативна пам'ять не повинна перевищувати 2 ГБ на кожну працюючу одиницю сервісу.

3.1.3 Визначення головних компонентів комплексу ПЗ.

Спираючись на визначені вимоги до функціональності, було створено представлення головних компонентів цільової системи інструментарію та сервісів:

- Identity & Access Management Service (IAM) - сервіс управління обліковими записами та доступом до ресурсів.

  Мета: забезпечення уніфікованого контролю за доступом до ресурсів та сервісів у системі, включно з рівнями прав доступу (RBAC) та додатково залученням особливої конфігурації для критичних ресурсів, такі як управління структурою проекта або розгортання/вилучення активних сервісів (MAC) [19].

  Вимоги до забезпечення роботи: наявна реляційна БД для збереження даних про облікові записи та дій, що виконуються щодо них (дані аудиту).

  Особливість системи IAM у даній системи складається не лише у використанні стандартизованих протоколів авторизації та аутентифікації OAuth 2.0/OpenID Connect, а й залучення додаткових механізмів безпеки, такі як багатофакторна аутентифікація за допомогою секретного коду або NFC картки, що є унікальними для кожного облікового запису. Цей метод застосовується не лише для початкового підтвердження спроби входу у систему, а й для виконання критичних операцій.

  Архітектура компоненту, а саме використання єдиного API для всіх клієнтських додатків повинна дозволяти інтеграцію майже будь-якого сервісу або інструменту з використанням бібліотек авторизації для C# або TypeScript.

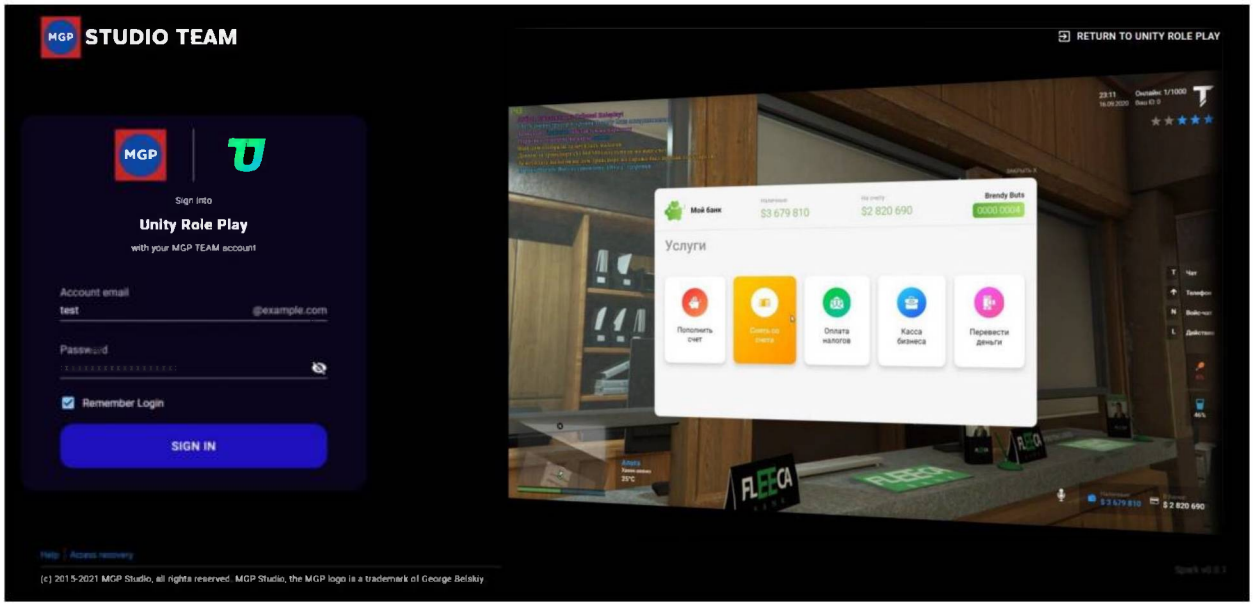

Рисунок 3.1 – приклад застосування сервісу IAM для авторизації у цільовому сервісі за допомогою веб-інтерфейсу

- Project Definition & Management Service (PDMS) – сервіс визначення та управління проектами програмних продуктів.

  Мета: стандартизація вимог, метаданих, компонентів, задач робіт та інформації щодо програмного продукту у єдину групу (проект), надання можливостей щодо динамічного управління та зміни більшості параметрів та інтеграція до іншого інструментального ПЗ.

  Вимоги до забезпечення роботи: наявна реляційна БД для збереження даних щодо проекту та доступу до них через API інтегрованими інструментами та сервісами.

  Представляє собою API сервіс, що надає доступ до інформації щодо проектів та можливість проведення операцій будь-якими сервісами або інструментами, включеними у робочий процес, таким чином об’єднуючи різних членів команди та надаючи їм можливість синхронізації не лише щодо стану робіт, а й для об’єднаного тестування.
- Desktop Hub – допоміжний додаток для ПК, що надає швидкий доступ до робочого середовища.

Слугує спрощеним методом доступу до всіх пов’язаних з одним або багатьма проектами інструментів та сервісів, дозволяючи роботу над будь-якою ревізією коду, медіаконтенту, задіювати різне програмне забезпечення з інтеграцією проектної інформації, без додаткового налаштування та необхідності оновлювати статус вручну. При роботі без підключення до сервера, додаток має змогу надавати кешовані дані щодо проекту та зберігати останні зміни для їх подальшої синхронізації з головним сервісом для інших членів команди.

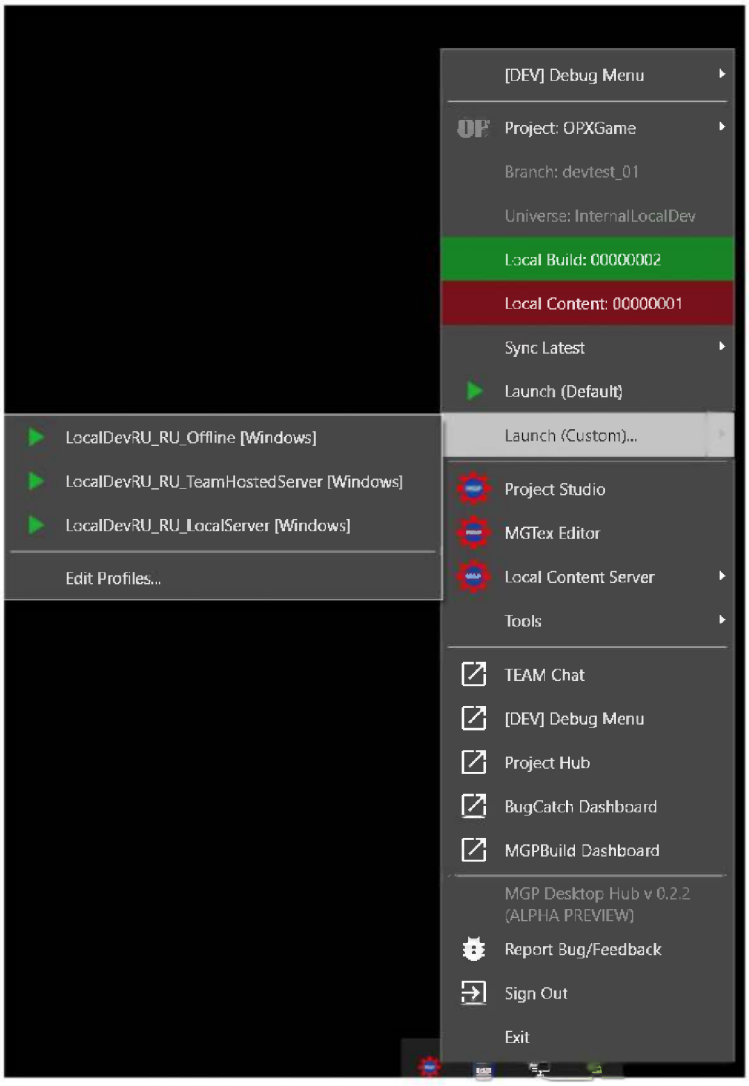

Рисунок 3.2 – приклад інтеграції додатку Desktop Hub у робоче середовище з обраним проектом (включаючи конфігурації для запуску, інтегровані інструменти та інші можливості)

- Operations Management Service – сервіс для централізованого управління та контролю за операціями та роботою компонентів програмних продуктів (як зовнішних, так і внутрішніх).

  Мета: надання централізованого доступу до всіх можливих аспектів роботи сервісної частини комплексних програмних продуктів, забезпечення динамічного внесення змін та огляду статистичних даних для формування уявлення про статус роботи ПЗ та можливі подальші дії. Цей сервіс є одним з найбільших компонентів комплексу як по складу, так і по функціоналу, включаючи у себе веб-додаток, внутрішні сервіси для забезпечення підтримки системи віртуальних мереж, а також централізованого розгортання та роботи зі сервісами (використовуючи окремі сервіси для моніторингу та управління цільовими, так звані “агенти” та “хост-менеджери”).

  Як і всі інші компоненти ПЗ, цей сервіс надає API для інтеграції до інших інструментів або окремих систем, наприклад для збору та відображенні статистики на сторінках статусу, чи швидкого управління через додаток Desktop Hub.

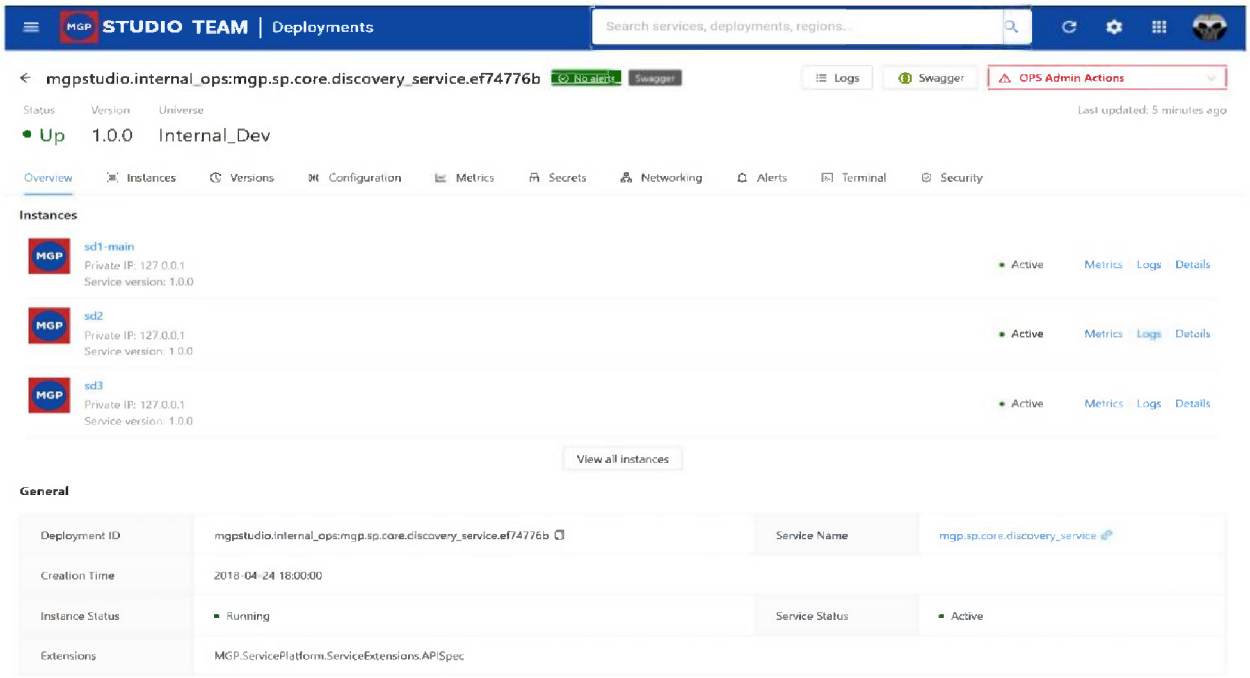

Рисунок 3.3 – приклад застосування веб-додатку сервісу для перегляду інформації щодо працюючого внутрішнього сервісу, включно з інформацією про сервіс та можливими операціями щодо нього

3.1.4 Визначення основних елементарних логічних блоків.

Через побудову більшості інструментального та сервісного ПЗ на спільній кодовій базі, має сенс виведення елементів коду, що використовуються у багатьох частинах до окремих елементів. Використання принципу DRY (“Don’t Repeat Yourself”, “не повторюй себе”) забезпечує наявність єдиної реалізації спільних функцій, особливо базових. До таких функцій та можливостей були віднесені наступні:

- Збір та обробка лог-даних, що включає у себе код, що відповідає за прийняття нових подій, їх обробку згідно з налаштованими правилами (видалення конфіденційної інформації та ін.), та відправка інформації щодо подій у відповідних форматах (JSON, простим текстом або у іншому вигляді) до відповідних приймачів (може бути як файлом, віддаленим сервером або іншим інструментом на локальному комп’ютері);

- Система Dependency Injection, що дозволяє реєструвати спільні класи програмного коду у контейнерах, та згодом використовувати функціонал таких класів за допомогою інтерфейсів у цільових класах/об'єктах таких класів;
- Зберігання згенерованих відповідно до мови програмування класів з моделями даних (детальніше про моделі даних у підрозділі 3.3);
- Спільні бібліотеки з допоміжним функціоналом (для роботи з рядками тексту, зчитування або запису даних різного типу та інше);
- Для сервісів – окрема бібліотека длі забезпечення функціональності Service Discovery;
- Для інструментального ПЗ на ПК – спіьні бібліотеки для інтерфейсу користувача.

3.1.5 Загальна архітектурна схема моделі комплексу ПЗ.

Схематичне представлення проекту моделі системи інструментального та сервісного ПЗ є набором окремих схем, що відображають структуру та відношення як між основними компонентами, так і окремо для кожного компоненту. Модель була спроектована у різних видах представлень, включаючи архітектурне, моделей даних та їх відношень, діаграми послідовності для основних алгоритмів логіки окремих компонентів, та об'єктне.

Схемою у додатку Е представлено загальну архітектуру моделі косплексу ПЗ на компонентному рівні (окреме виділення основних сервісів та клієнтів, а також шляхів їх взаємодії).

3.1.6 Визначення можливостей розширюваності для застосунків.

Підтримка динамічного та універсального методу розширення функціоналу є ключовим для реалізації вимог до проекту щодо незалежності та інтеграції компонентів.

Для різних категорій компонентів повинні застосовуватися відповідні методи та моделі розширень, відповідно до архітектури ПЗ, обмежень платформи та необхідних цілей від додавання нових елементів.

Загалом, типи розширення у комплексі ПЗ поділяються на наступні:

- Розширення функціоналу, що включає додавання нових віджетів інтерфейсу, або нової бізнес-логіки (як до інструментів, так і до сервісів);
- Динамічна інтеграція сервісів, з використанням методів Service Discovery та стандартних API для забезпечення універсальної взаємодії незалежно від типу або розташування сервісу;
- Підтримка розширення без необхідності перезапуску ПЗ, що є дуже важливим для активних сервісів, які можуть обслуговувати дуже багато запитів одночасно;
- Залучення динамічної конфігурації та налаштувань відповідно до зовнішнього середовища для автоматизованого додавання або вилучення розширень з системи (як на клієнтській, так і сервісній сторонах).

3.2 Основні підходи, методи проектування та інтеграції компонентів.

Для забезпечення виконання вимог щодо інтегрованої роботи набору інструментів та сервісів для надання спільних можливостей для ведення процесів розробки та проектування, потрібно визначити основні технічні складові щодо проектування елементів системи та їх інтеграції та взаємозв’язків.

3.2.1 Підтримка крос-платформної роботи.

Забезпечення крос-платформної роботи є однією з ключових вимог до проекту, як для клієнтського ПЗ, так і сервісного. На даний час спостерігається тренд розробки інструментів розробки, що підтримують лише операційну систему Windows, або сервісів, що працюють лише на архітектурі x64, що не відповідає поставленим мінімальним системним вимогам для цільової системи.

Для досягнення портативності постає мета вибору основи, що забезпечить виконання поставленої вимоги без додаткових витрат матеріальних ресурсів, або часу, та дозволить зберігти суміжність логічних блоків та програмного коду між різним ПЗ комплексу. Такою основою є мова програмування та засоби виконання зібраного програмного коду.

Для порівняння було обрано 3 найпопулярніші: C++, C# та Java. Критерії порівняння встановлені наступними:

- Метод виконання (інтерпретація або виконання компільованого машинного коду);
- Залежність переносимості готової програми від інструментів компіляції;
- Залежність стандартних бібліотек мови програмування від платформи;
- Відносна швидкість виконання коду;
- Легкість у налаштуванні середи розробки під інструменти та мову програмування.

Результати порівняння представлені у таблиці 3.1. Остаточний вибір мови програмування зроблений за отриманими результатами.

Таблиця 3.1

Порівняльна таблиця мов програмування за критеріями портативності

Критерій	C++	C#	Java
Метод виконання	Компіляція до машинного коду	Компіляція до проміжкового коду та виконання інтерпритатором	Компіляція до проміжкового байт-коду та виконання інтерпритатором віртуальної машини (JVM)
Залежність від інструментів	Так (відмінність компіляторів для кожної платформи)	Ні (однаковий процес компіляції на всіх платформах)	Ні (однаковий процес компіляції на всіх платформах)
Залежність від адаптації до платформ	Так (відмінні бібліотеки та API платформ без абстракції)	Мінімальна (наявні абстракції API платформ для крос-платформного використання, за наявності)	Мінімальна (наявні абстракції API платформ для більшості стандартних можливостей)
Відносна швидкість виконання коду [20]	Найшвидша	Швидка	Менш швидка
Легкість налаштування середовища	Найскладніша (велика кількість різноманітних інструментів та типів включення бібліотек)	Найпростіша (однаковий набір інструментів та процес включення бібліотек до проекту)	Складна (велика кількість бібліотек, що не підтримують останню версію Java, а також багато версій самого середовища виконання)

Примітки:

- Для C++
  - Найскладніше налаштування середовища зумовлено розбіжністю у прийнятті стандартів включення сторонніх

бібліотек до проектів (статичне, динамічне включення, додавання header-файлів напряму у код), а також алгоритмами збірки проектів (з застосуванням CMake або інших інструментів);

- Через історичний вибір при проектуванні та розробці різних ОС, стадартом став C++, але з цим виникла проблема різниці у API через відмінності у функціональності та архітектурі систем. Відповідно до цих рішень сформовані бібліотеки, що надають доступ до системних API також є унікальними до кожної платформи, а відсутність абстракцій створює проблему – баланс між внесенням декількох варіантів виконання подібних дій для підтримки багатьох платформ одночасно, та ризиком проблем розуміння коду (зазвичай, визначення хост-системи виконується через використання спеціальних блоків коду “#ifdef”, що мають обмеження щодо місць використання).

- Для C#
  - Перевагою мови серед інших є застосування єдиної моделі включення сторонніх бібліотек до проекту, що стала стандартом серед розробників, та є незалежною від платформи [21];
  - Відкритий код компілятору Roslyn та середи виконання [22] забезпечує більш швидкий розвиток мови програмування та платформи, а офіційна підтримка Microsoft для багатьох популярних платформ робить ідеальним вибір C# для розробки проекту даної роботи.

Окремо від обрання бази для програмного коду є вибір способу представлення інформації до користувачів та надання їм можливостей по взаємодії, а саме – основи для інтерфейсу (UI), включаючи його відображення,

обробку подій, а також набір стандартних компонентів (віджетів). Через значні відмінності у реалізації графічного інтерфейсу на платформах, використання єдиного фреймворку для індивідуальних додатків поза веб-браузером є проблематичним.

Для платформ iOS та Android найкращим є застосування вбудованих засобів, але використання їх у мові C# не є доступним через орієнтацію розробки на мови Swift та Kotlin (Java). Вирішенням даної проблеми є використання бібліотеки Xamarin, мета якої – надання доступу до можливостей платформ та забезпечення роботи програм, написаних на C#, на даних мобільних платформах [23].

Для комп'ютерних ОС постає більш складний вибір, тому що навіть за (теоретичної) підтримки використання Xamarin як крос-платформного рішення, воно орієнтується на використання бібліотек UI, що специфічні до кожної платформи, та виступає лише "обгорткою" до них, тому застосування розширених можливостей або не є можливим, або може призвести до критичних проблем у зв'язку з неповноцінністю розробки [24, 25].

Серед інших наявних бібліотек – WPF та GTK# – є наявною основна проблема – прив'язка до платформи, тобто функціональні можливості кожної бібліотеки обмежені платформою, для якої вони були розроблені (WPF для Windows та GTK# для Linux), що унеможливлює їх застосування як крос-платформного рішення для інтерфейсу інструментів [26].

Вибором стала бібліотека Avalonia через наступні причини:

- Відкритий програмний код та ліцензія MIT, що дозволяє вносити довільні зміни для локального використання та застосовувати бібліотеку у комерційних продуктах;
- Використання мови XAML як одного з варіантів проектування схем інтерфейсу вікон та елементів, що є подібним до WPF, тому є дуже простим для переходу від однієї бібліотеки до нової, а також включає доповнення, такі як система стилів;

- Працює на всіх трьох популярних платформах на ПК – Windows, macOS, та Linux, через використання власного відображення графічної складової інтерфейсу, що допомагає зберігати однаковий дизайн;
- Має вбудований інструмент для огляду дерева інтерфейсних елементів та окремих властивостей кожного під час роботи програми, що значно спрощує відладку та проектування;
- Завдяки використанню ресурсів GPU, фреймворк Avalonia забезпечує високу продуктивність, що є особливо важливим для комплексних інструментів.

У якості бази для веб-додатків було обрано TypeScript (мова, що надбудована над мовою Javascript, з підтримкою суворої типізації), а для інтерфейсу користувача – бібліотеку React, особливістю якої є використання механізму “dirty state” для динамічної зміни елементів інтерфейсу на веб-сторінці, але лише тих, для яких змінився стан. У React такими основними елементами є так звані “компоненти”, що можуть включати у собі один або більше стандартних елементів мови HTML, а також окрему логіку та стан даних, що використовується тим чи іншим компонентом [27].

3.2.2 Оптимізація функціоналу та інтерфейсу – критерії та методи.

Щодо вибору основи для будування інетрументів – основним питанням є вибір між веб-додатком або “нативним” для будування інструментального ПЗ. Цей вибір залежить від цільової функціональності та вимог до компоненту системи. Загалом, для кожного клієнтського компоненту постають наступні питання:

- Пріорітет на швидкість відгуку, чи на адаптивність інтерфейсу до різних форм екранів пристроїв?
- Чи є необхідність у використанні специфічних функцій окремих платформ?

- Чи є необхідність у роботі з файлами у локальному сховищі?
- Чи потребує додаток окремого представлення, або об'єднання з використанням спільних інтерфейсних елементів з іншими, у тому числі в плані дизайну?

Використання конкретних методів є специфічним для кожної частини комплексу ПЗ у даному проекті, але з загальних можливих методів можна виділити наступні:

- Використання спільних дизайн-систем для інтерфейсів користувача. Такий підхід є одним з найбільш ефективних для формування спільного дизайну для деіклькох клієнтів, тому що дозволяє формувати спільний дизайн віджетів, затверджувати стандартні ресурси (шрифти, кольори), розмірність елементів на екрані, у тому числі відносно інших, а також теми оформлення. Відповідно до дизайн систем розробниками формується технічна склдова інтерфейсу, реалізація основних частин, у тому числі управління темами, постачальники ресурсів, логіка віджетів, їх параметри, залежність та можливості взаємодії, та інше;
- Подібно до UI, створення спільних елементів логіки є також необхідним для проектування спільних або незалежних інтегрованих програмних інструментів, включаючи базову логіку та функціональну, відповідно до вимог. Існують рекомендації для виділення спільної логіки до окремих бібліотек/модулів, що може бути корисним для збереження залежностей між інструментом та специфічною версією спільної бібліотеки, для якої у наступній може бути змінена логіка, що не є підтриманим у самому інструменті, тобто зберігається як модульність, так і підтримка розбіжності версій, якщо це потрібно.

3.2.3 Спільні моделі даних.

Відповідно до проектування спільних функціональних та технологічних елементів проекту, формування спільних моделей даних є дуже важливим процесом, тому що для інтеграції багатьох компонентів та їх успішної взаємодії потребується розуміння залежностей між різними моделями та їх спільне використання різними сервісами та іншими компонентами під час обміну даними.

Для прикладу можна привести клієнт-сервісну взаємодію, коли обидві сторони повинні мати представлення про структури інформаційних блоків, з якими проводяться спільні операції. У таких випадках, навіть для незалежних на структурному рівні компонентів є необхідною залежність на рівні моделей даних. Однак, існують виключення.

Для забезпечення обміну даними між двома категоріями програмного забезпечення не є необхідним наявність оригінальних моделей даних, а лише потрібних частин для окремих операцій. Такими проміжними моделями слугують так звані DTO (Data Transfer Object), тобто тимчасові об’єкти, що включають елементи оригінальної моделі даних, але не є оригінальним представленням через спеціалізацію для використання у конкретних операціях. На рисунку 3.4 відображено приклад таких операцій та моделей даних. У випадку з моделлю ServiceConfig, ServiceConfigDto є тимчасовим представленням оригінальної моделі, що видається у відповідь на запит /services/details. Та ж тимчасова модель включає у себе 2 інші моделі, що також є відображеннями включених до оригінальної моделей.

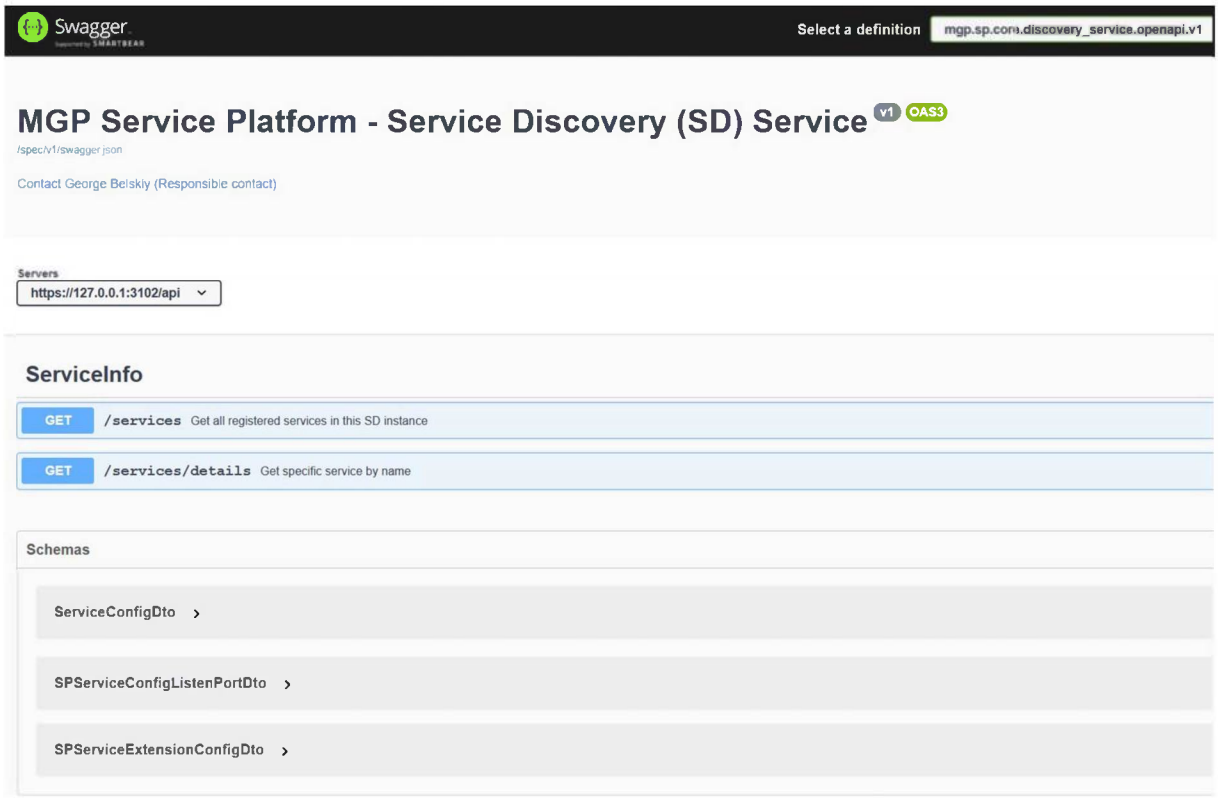

Рисунок 3.4 – приклад використання концепту DTO при формуванні індивідуальних представлень спільних моделей даних, залежних від операцій

3.2.4 Засоби комунікації між компонентами.

При використанні мікросервісної архітектури дуже важливо забезпечення як незалежної роботи окремих сервісів, так і єдиний спосіб взаємодії між ними, обміну даними та управління. Додатково повинні бути встановлені методи комунікації між сервісними та клієнтськими частинами ПЗ.

- Взаємодія клієнт-сервіс – HTTPS (REST). Цей протокол є стандартом в технологічній індустрії та є традиційним при застосуванні для надання API методів. Використання принципу REST забезпечує розділення можливих операцій з даними за допомогою типів запитів HTTP та статичне визначення типів даних на обох сторонах [28];
- Міжсервісна взаємодія – gRPC. Технологія gRPC є розвитком моделі RPC (віддалений виклик процедур) від Google, де для

обміну повідомленнями застосовується мова визначення схем (DSL) під назвою Protocol Buffers (Protobuf). Додатково, за допомогою Protobuf та інструментів генерації коду можливо спроектувати схему сервісних функцій та згенерувати готовий програмний код, до якого лише потребується додати необхідну логіку. Переваги Protobuf та gRPC є у економії часу (немає необхідності застосовувати ручне написання службового коду), спільному визначенні інтерфейсів та моделей даних для різних сервісів та підтримці багатьох мов програмування для автоматизованої генерації службового коду [29].

3.3 Збереження даних

Для комплексної системи, що об'єднує багато різноманітних типів програмного забезпечення постає питання управління різними типами даних – як у пам'яті під час роботи, так і при збереженні. Особливо важливим є обрання найкращих методів згідно не лише області, до якої відноситься компонент ПЗ (клієнт, чи сервер), а й ситуації, при якій проходить обмін або обробка даних. Також є важливим прийняття до уваги проектування самих моделей даних та використання інструментів, що допоможуть зберегти час при впровадженні моделей у код як структури для зберігання у файловій системі, БД, а також для обробки.

3.3.1 Формування моделей, визначення основних дійових осіб.

При формуванні моделей та представлень даних потрібно брати до уваги наступні критичні моменти:

- Призначення структури – допоміжна, чи операційна, тобто є модель самодостатньою одиницею, що застосовується для обробки та у проведенні операцій, чи є одним зі службових типів (такі як перерахування, набір ключ-значень з статичними

значеннями, тощо). Відповідно до обраної категорії потребується встановлення унікальних ідентифікаторів у моделі для полегшення індексації та доступу до різних значень на базі однакової моделі (такі як інформація про аккаунт користувача);

- Виділення комплексного набору інформації щодо дійової особи або об'єкту системи у окремі, залежні моделі – при описі складного об'єкту або особи, що потребує вказання великої кількості параметрів у модельному представленні, потрібно застосовувати розділення параметрів на дві, чи більше моделей, що включаються один до одної, а "корнем" такого дерева становиться основна модель об'єкту чи особи. Зазвичай, поділення параметрів виконується за його смислового відношення до інших, таким чином відносячи сміжні параметри до окремих груп. У той же самий час, групи параметрів, що за значенням є однаковими для декількох осіб чи об'єктів можуть бути винисені у окремі спільні моделі;
- Різні типи сховищ та серед обробки даних потребують адаптацій параметрів або відношень моделі – особливо це стосується зберігання складних структур даних у БД або обмін даними між двома компонентами (наприклад, клієнт та сервіс), де для операції потребуються не всі параметри оригінальної моделі. Такі "тимчасові" моделей можуть також включати відношення між ідентифікаторами об'єктів моделей, що зазвичай застосовується при збереженні у БД міжмодельних відошень для випадків "багато-до-багато", або "один-до-багато".

Одним з основних кроків при проектуванні моделей даних для косплексної системи є визначення предметної області системи, дійових осіб що з нею взаємодіють, та основних об'єктів, що використовуються у системі,

а також параметрів осіб та об’єктів. Відповідно до опису системи, та функціональних вимог, можна виділити наступні:

- Облікові записи користувачів
  - унікальний ідентифікатор,
  - загальний рівень доступу (адміністратор, користувач),
  - статус аккаунта (заблокований, підтверджений),
  - дата реєстрації,
- Конфігурація аккаунтів для входу у систему
  - відносний ідентифікатор аккаунта,
  - облікові дані (тип, значення, дата останніх змін; включаюч загальні типи, такі як електронна пошта, логін, пароль),
  - налаштування двох-факторної аутентифікації (статус активності, зареєстровані мобільні додатки для підтвердження, хеш секретного коду),
- Інформація аудиту щодо дій у аккаунтах
  - тип дії, виконаної щодо обліково запису,
  - джерело запиту на виконання (користувач, адміністратор, або ресурс системи, включаючи який саме ресурс),
  - час виконання,
  - додаткова інформація (може включати значення налаштувань, які були змінені),
- Під’єднані профілі сервісів (параметри залежні від цільового сервісу)
  - ідентифікатор сервісного проекту (для PDMS та системи авторизації),
  - тип профілю (обмежений для одного сервісу, або спільний та доступний для будь-якого),
- Активні сесії авторизації
  - ідентифікатор сесії,

  - дата створення,
  - дата останньої активності користувача,
  - ресурс системи, для якого була створена сесія (сервіс або клієнт),
  - розташування, з якого була створена сессія (місто та країна),
  - токен аутентифікації (унікальний для сесії),
  - видані дозволи від користувача, що доступні для ресурсу,
- Клієнт цільового ресурсу для сервісу авторизації,
  - унікальний ідентифікатор,
  - кодова назва,
  - логотип,
  - назва для відображення,
  - необхідні дозволи для доступу до операцій або даних,
  - загальні політики авторизації (необхідність 2FA, входу перед початком кожної нової сесії, міімальна кількість символів у паролі, тощо),
  - унікальні політики для ресурсу (наприклад, необхідність користувача бути учасником групи),
- Організаційні групи
  - унікальний ідентифікатор,
  - дата створення,
  - інформація аудиту (сміжні параметри до облікових записів),
  - учасники групи,
  - назва,
  - логотип,
  - права доступу користувачів до різних операцій та ресурсів у PDMS,
  - проекти

- Інформація щодо проекту програмного продукту
    - унікальний ідентифікатор,
    - тип проекту (доступний для усії користувачів системи, обмежений до групи, або окремих користувачів групи),
    - назва,
    - кодова назва,
    - логотип,
    - опис,
    - дата створення,
    - інформація аудиту (сміжні параметри до облікових записів),
    - список програмних модулів програмного продукту,
    - інтеграції
        - тип інтеграції (документація, пакети ПЗ, система авторизації/реєстрації, CI/CD, та інші),
        - статус активності,
        - унікальні ідентифікатори до даних, що надаються інтеграціями,
- Модулі програмного продукту
    - унікальний ідентифікатор (у рамках одного проекту)
    - кодова назва,
    - список “функцій” модулю,
    - цільова архітектура та програмна платформа,
    - список включених “функціональних” блоків (спільні елементи ПЗ, що можуть бути використані у декількох проектах або модулях),
- Функціональний блок ПЗ програмного продукту
    - унікальний ідентифікатор (у рамках системи загалом),

  - список підтримуваних цільових архітектур та програмних платформ систем,
  - тип блоку (бібліотека, викоуваний файл, ресурс даних, клієнтська бібліотека сервісу),
  - список параметрів типу “ключ-значення”, унікальних для типу блоку, якщо застосовуються,
- Сервіс
  - унікальний ідентифікатор (для реєстру сервісів)
  - кодова назва,
  - дата створення,
  - остання версія,
  - список усіх версій та дата їх розгортання (або відсутність дати, якщо розгортання не проводилось),
  - інтегровані розширення,
- Розгортання сервісу
  - унікальний ідентифікатор,
  - версія розгортання,
  - дата розгортання,
  - прив’язаний ідентифікатор сервісу, що розгортався на робочі одиниці,
  - кількість робочих одиниць,
  - приналежність до віртуальної мережі,
  - встановлені політики взаємодії з іншими сервісами (вхідний/вихідний трафік від та до зареєстрованих портів сервісу до яких саме цільових сервісів, тип дозволеного трафіку, максимальна пропускна спроможність, дозволений період активності та інші),
  - стан активності інтегрованих розширень,
- Робоча одиниця розгортання сервісу

- адреса (для реальної та віртуальної мереж),
- список відповідностей внутрішніх та зовнішних портів (внутрішні застосовуються у контейнері та є налаштованими у ПЗ сервісу, зовнішні налаштовані конфігурацією та використовуються для взаємодії з сервісом),
- лог-дані,
- статистичні метрики,
- кодова назва,
- статус активності.

Представлені об'єкти та їх властивості визначають деякі з понад 200 індивідуальних моделей даних, що застосовуються у проекті, включаючи лише основі компоненти.

3.3.2 Розподілення відношень моделей до компонентів системи.

Прив'язка моделей даних до їх "власників", тобто компонентів, для яких вони є основними, виконується за допомогою встановлення відношення складу моделі та об'єкту, що вона визначає, до функціональних вимог кожного компоненту системи.

Використовуючи даний принцип, основні об'єкти та особи, задіяні у системі, та їх моделі даних були розподілені відносно сервісів за наступним чином:

- IAM – облікові дані користувача, конфігурація для входу, інформація аудиту аккаунтів, під'єднані профілі користувачів, активні сесії аутентифікації, опис клієнтів ресурсів для авторизації;
- PDMS – організаційні групи, проект програмного продукту та його метадані, модулі та функціональні блоки програмного продукту;
- OPS – Сервіс, розгортання сервісу та робочі одиниці розгортань.

Додаткові моделі даних, що входять у склад основних, або службові, відносяться до відповідних компонентів, що й основні моделі.

3.3.3 Засоби для збереження та їх цільове призначення.

З основних базових засобів збереження даних виділено наступні:

- Файлова система програмної платформи – надається окремо кожною ОС та слугує для збереження даних як окремі файли на диску для довготривалого тримання. Основним недоліком роботи з таким засобом є апаратні обмеження сховищ даних, тому що вони є зазвичай найповільнішими, та рекомендуються для використання лише у ситуаціях, де затримка від очікування запису або читання не є критичною. Такий тип зберігання підходить для довільних типів інформації;
- Реляційна БД – є пріоритетним засобом збереження структуроваих даних, визначених статичними моделями. Застосування реляційної БД зумовлено тим, що між різними моделями даних є відносини, особливо коли одна комплексна модель включає декілька інших. Такі типи БД забезпечують більш швидку роботу з відносними операціями щодо малих та великих обсягів даних, а також підтримують режим транзакцій, тобто виконання атомарних операцій, що включають одну або більше команди SQL, при помилці одної з яких вся операція відміняється. Через збереження даних на файловій системі робота з БД у реальному часі також не рекомендується;
- Сервіси кешування – розроблені для вирішення проблеми роботи з даними у реальному часі, дозволяючи сервісам отримувати доступ до даних, що раніше використовувалися. Завдяки API функціям сервіс має змогу записувати різні блоки даних до кешуючого сервісу, а згодом, на 2 ті інші рази отримувати існуючу

копію, не проводячи знову довготривалих операцій з структурою даних, що дуже заощаджує час та зменшує затримку. Додатково затримку зменшує і принцип роботи кешу, тому що всі дані зберігаються у оперативній пам'яті, що є на порядок швидшою за постійне сховище.

3.3.4 Причини та переваги вибору формату Protobuf.

Protobuf – мова визначення схем даних від Google. Основними елементами є так звані "повідомлення", що представляють собою структури даних. Назва повідомленнями походить від цільового застосування – обмін повідомленнями між декількох ресурсів. Для розуміння переваг вибору саме мови та інструментів Protobuf, у таблиці 3.2 наведені критерії та приклади інших систем серіалізації даних:

Таблиця 3.2

Порівняльна таблиця систем серіалізації даних

Критерій	Protobuf	MessagePack	JSON
Тип серіалізації	бінарний	бінарний	текст
Визначення моделі	файл схеми	у програмному коді	JSON Schema (окремий текстовий блок)
Підтримка змін моделі	+	+	+
Засоби серіалізації	надаються бібліотекою	надаються бібліотекою	надаються стандартною бібліотекою мови програмування/вручну
Портативність	Обмежена підтримкою компілятора схеми у код	Обмежена наявністю реалізації	Необмежена, підтримує обробку як текст (розділення, зливання, тощо)
Складність використання	складна	складна	найпростіша
Відносна швидкість роботи [30]	найнижча	висока	найвища

Таблиця 3.2 (продовження)

Критерій	Protobuf	MessagePack	JSON
Валідація схеми	+	-	- (опціонально у окремих випадках)
Відносний розмір вихідних даних [30]	найнижчий	низький	найвищий

З отриманих результатів дослідження можна встановити, що навіть при оптимальних результатах роботи, система MessagePack не є ідеальним кандидатом через складність використання у плані проектування спільних моделей даних для різних мов програмування та багатьох програмних компонентів. На практиці, при застосуванні асинхронного підходу при взаємодії та обміні даними, затримки при серіалізації Protobuf можуть бути або майже непомітними, або не впливати на загальний процес користування ПЗ.

3.3.5 SQL проти NoSQL – обгрунтування вибору, порівняння систем.

На даний час, основними архітектурними моделями, що реалізовані у існуючих базах даних, є наступні:

- реляційна (SQL),
- нереляційна (NoSQL).

Основними відмінностями між двома моделями є наступні [31, 32]:

- СУБД на базі SQL використовують визначену завчасно структуру даних, що є статичною та не може бути змінена (для більшості випадків);
- СУБД на базі NoSQL можуть представляти собою різноманітні типи структур даних, включаючи документну, ключ-значеня, та графову, забезпечуючи зберігання структур, що залежать від моделей даних, а не обмежень технологічної бази ПЗ;

- SQL є більш оптимальним вибором для роботи зі складними запитами та операціями щодо даних, які можуть об'єднувати різні структури (таблиці) одночасно;
- SQL є де-факто стандартом індустрії, що має багато реалізацій та використовується на великій кількості складних проектів з мільйонами користувачів [32];
- Більшість БД на основі SQL підтримують мехінізм транзакцій, що дозволяє виконувати пакетні запити, включаючи залежні один від одного, з гарантією виконання усіх або жодного, що забезпечує цілісність та постійність даних. Цим функціоналом також підтримується виконання принципу ACID;
- Кластеризація, тобто об'єднання одиниць БД у єдину систему за принципом "головний-залежний".

У якості базової архітектури для баз даних основних компонентів ПЗ було обрано SQL через наступні причини:

- Необхідність використання транзакційних операцій зумовлена комплексними сервісними компонентами, що можуть проводити дії над великою кількістю залежних між собою структур даних;
- Наявність бібліотек ORM для мов програмування, що забезпечують взаємодію з БД не зважаючи на конкретну реалізацію;
- Можливість використання вбудованих операцій у СУБД для розділення відповідальності між бізнес-логікою та управлінням збереження даних, а також забезпечення більш високого рівня безпеки через виключення можливості відправки зловмисних запитів то СУБД;
- Більша кількість реалізацій, більша кількість сумісних платформ та архітектур процесорів.

Окремо, причиною відхилення графової моделі з можливих варіантів, не зважаючи на можливість динамічно змінювати структуру та встановлювати більш є відносна складність їх впровадження, відсутність підтримки використання транзакційних операцій, відсутність необхідності проведення додаткової обробки даних у БД (для огляду відношень та інших типів аналізу), а також через відносну статичність моделей даних на протязі часу [33], роблячи такий варіант відносно складним.

3.4 Огляд структури сервісних (backend) компонентів.

Архітектурна модель сервісних компонентів включає у себе елементи та методи реалізації функціональності та взаємного сполучення і взаємодії сервісів, підтримку комплексних структур складних сервісних підсистем, а також спільні методи для управління взаємодією з іншими сервісами та клієнтами, що спрямовані на забезпечення безпеки, цілісності, безперервності роботи, та динамічності у мікросервісній моделі даного проекту.

3.4.1 Основні спільні технологічні та логічні частини.

Основне сервісне ПЗ комплексу побудоване на мові програмування C#. Додатково до використання загальних спільних лоігчних елементів, описаних у п. 3.1.4, сервісне програмне забезпечення будується згідно вимог, тому є необхідність у впровадженні наступних можливостей, що є спільними для сервісної частини комплексу ПЗ загалом:

- Service Discovery – забезпечення динамічного виявлення та ідентифікації сервісів у мікросервісній реалізації.

  Через застосування принципу незалежності та наявним вимогам щодо підтримки інтеграції між сервісами різних типів, постає необхідність у забпзепеченні динамічного контролю та управління інформацією щодо сервісів. Основними чинниками для впровадження даної можливості є наступні [34]:

- динамічна зміна кількості та типів активних сервісів,
- надання сервісами відкритих API точок (в окремих випадках більш ніж одної),
- динамічність розміщення сервісного ПЗ на системах (наприклад, без гарантій статичних IP адрес або портів).

За принципом впровадження, існують 3 головні необхідні частини:

- Реєстр сервісів, що контролюється або централізовано, або спільно кожною одиницею сервісів;
- Самі сервіси, що виконують завдання реєстрації інформації щодо них (назва, вид, версія та інше) та можливостей доступу (IP, порт, протокол, та інші параметри розташування) у реєстрі.

  Міжсервісна комунікація виконується через запит до реєстру щодо необхідного сервісу, та при отриманні інформації щодо доступу сервіс може проводити запити до цільового сервісу. Додатково, для забезпечення постійності та достовірності стану мікросервісної середи, реєстр має повідомляти інші сервіси про оновлення стану, наприклад при вилученні активної одиниці сервісу (деактивації, як запланованої, так і можливої аварійної), або додаванні нової;
- API роутер, що виконує передачу запитів від клієнтів до необхідних сервісів, спираючись на реєстр, або відповідає помилкою, якщо сервіс недоступний. Сам клієнт не взаємодіє напряму ні з реєстром, ні з індивідуальними сервісами.

- Стандартизація міжсервісної взаємодії за допомогою так званих “розширень” – єдиних моделей запитів та методів взаємодії, що відносяться до одної чи іншої категорії. Застосування такого принципу надає наступні переваги:

- Універсальність для міжсервісної взаємодії та зовнішнього управління, незалежно від версій та архітектур самих сервісів;
- Завдяки розділенню таких кінцевих точок на категорії (що є відповідними до розширення) є можливість відділення опціональних частин та застосуваня лише для тих сервісів, де це потрібно (наприклад, API для роботи з даними лише для сервісів-провайдерів даних);
- Динамічність, що забезпечує реєстрацію інформації щодо розширень у реєстрі сервісів, у тому числі додавання нових розширень.

- Моніторинг статусу та статистичних метрик сервісів [35, 36]:

  Для забезпечення підтримки стабільності роботи системи у цілому та окремих сервісів, для сервісних компонентів є необхідним відстеження різноманітних показників, а також періодичне інформування сервісів контролю та управління про статус роботи (так зване “серцебиття”, або heartbeat). З основних статистичних даних, що можуть збиратися з сервісів, можна виділити наступні:
  - Споживання ресурсів (пам’яті ОЗУ та постійного сховища даних, процесора);
  - Кількість зовнішніх запитів за проміжок часу, типи відповідей (помилка, успіх, та інші);
  - Джерела зовнішніх запитів (IP адреси, типи клієнтів);
  - Інформація щодо міжсервісних запитів (кількість, типи відповідей, джерела та цільові сервіси);
  - Активні сессії з користувачами (якщо сервіс використовує облікові записи);
  - Кількість запитів до БД за проміжок часу, їх типи та результат виконання (помилка або успіх);

- Моніторинг повідомлень про помилки у реальному часі та реактивне управління сервісами: зважаючи на отриману інформацію щодо стану сервісу та окремо інформації щодо помилок, які можуть виникнути під час його роботи, сервіси управління та моніторингу, згідно налаштувань, можуть прийняти рішення щодо виведення одиниці сервісу з експлуатації, або навіть усього набору з одного сервісу. Залежно від налаштувань, додатково може бути виконаний перезапуск або розгортання нової одиниці, якщо є потреба через навантаження на існуючі одиниці. Подібні типи операцій виконуються сервісами компоненту OPS.
- Міжсервісне логування запитів [37]: слугує важливою частиною операцій моніторингу роботи сервісів, але у порівнянні з іншими діями, ці операції мають пріоритет на клієнтське ПЗ. У більшості випадків, запит від клієнта може проходити через більш ніж один сервіс, а тому, для відстеження його проходження від отримання до відправки відповіді, або прив'язки спільних параметрів, таких як авторизована сессія, застосовується встановлення універсального унікального коду (у форматі UUID) кожному запиту. Використовуючи цей постійний код, розробники мають можливість скласти інформацію з лог-файлів різних сервісів для повного розуміння ситуації саме для конкретного запиту.

На прикладі сервісного компоненту PDMS, (основна мета – управління проектами та їх інформацією) на рисунку 3.5 надається структурна схема розподілення та взаємодії мікросервісів та супутних частин, у тому числі з клієнтським ПЗ (компонентом Desktop Hub) саме для Service Discovery.

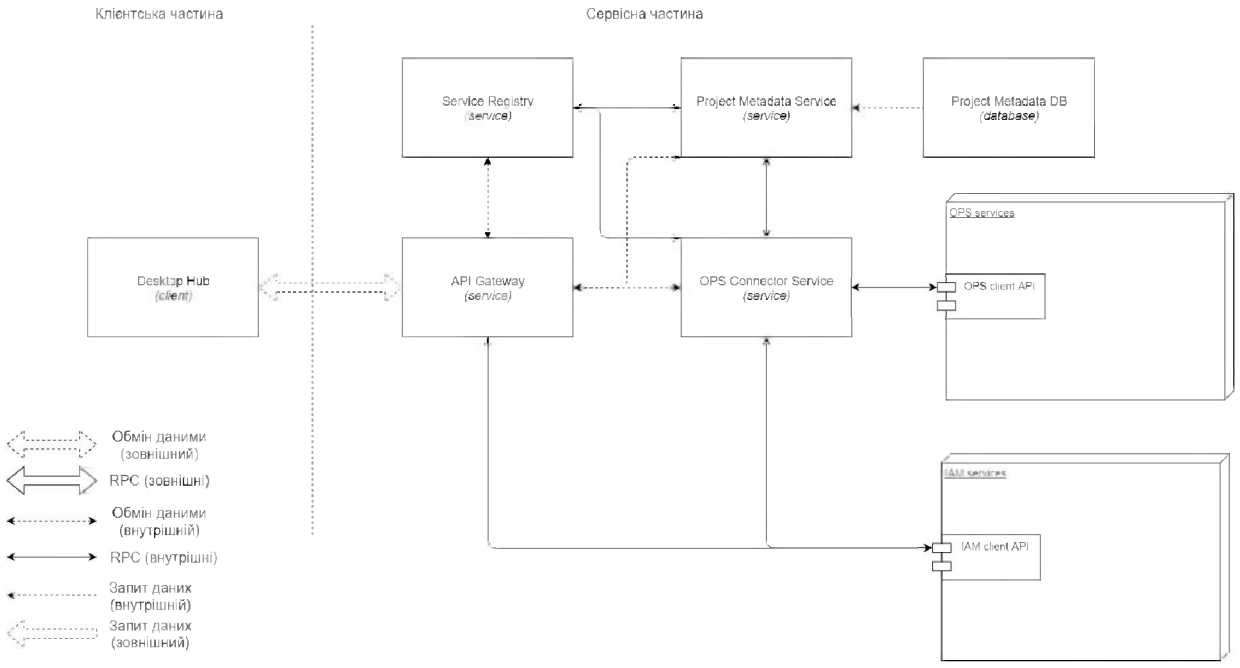

Рисунок 3.5 – схематичне відображення структурної схеми мікросервісного компоненту PDMS та сміжних компонентів архітектурної моделі прототипу комплексу ПЗ. До схеми включені додатково загальне представлення окремих сервісів, що використовуються сервісами компоненту PDMS (IAM та OPS).

На схемі відображено розділення на клієнтську та сервісну частину, а також різні типи взаємодії між компонентами, включаючи використання розширень, що надаються іншими сервісами для доступу до інформації щодо наявної авторизації (за допомогою сервісу IAM) та виконання операцій щодо розгортання сервісів цільового проекту (за допомогою сервісу OPS). API Gateway у свою чергу виступає так званим “пропускним пунктом” між клієнтами та сервісами, виконуючи одразу декілька важливих завдань:

- Перевірка наявної аутентифікації користувача у системі та авторизації (прав доступу) до сервісів компоненту PDMS;
- Перевірка встановлених політик обмеження, таких можливе як обмеження кількості запитів (rate limit), блокування адреси, з якої виконується запит (IP filtering), та інших;

- Ідентифікація цільового сервісу за допомогою розбіру запиту (використовуючи такі параметри, як URL, або інші) та пошуку у реєстрі сервісів;
- Інформування користувача про можливі помилки у запиті, відмову в обслуговуванні або інші проблеми на сервісній стороні (такі як відсутність активного цільового сервісу);
- Відокремлення логічно-незалежних частин системи (клієнтської від сервісної).

У додатку Ґ відображена діаграма послідовності обробки запиту від клієнту до одного з сервісів компоненту PDMS. Суть запиту – отримати інформацію щодо проекту. Початкові дані запиту включають цільовий URL, тип POST, а також уніальний ідентифікатор проекту.

3.4.2 Основні відмінності.

На прикладі сервісів компоненту OPS можна побачити основні відмінності на функціональному рівні. Проектування логіки та механізмів для даного компоненту є специфічним через саму мету застосування – управління іншими сервісами та їх розгортаннями, включно з сервісами цільових програмних продуктів та внутрішніх сервісів самого комплексу ПЗ, надаючи системі можливість самобазування та самоуправління (зі встановленими обмеженнями, щоб завадити ризику краху истеми через зняття розгортання самих основних сервісів, що відносяться до компонентів OPS або IAM).

На рисунку 3.6 відображено архітектурну схему сервісів компоненту OPS, а також внутрішньої та зовнішньої їх взаємодії (на прикладі компоненту Desktop Hub я клієнту).

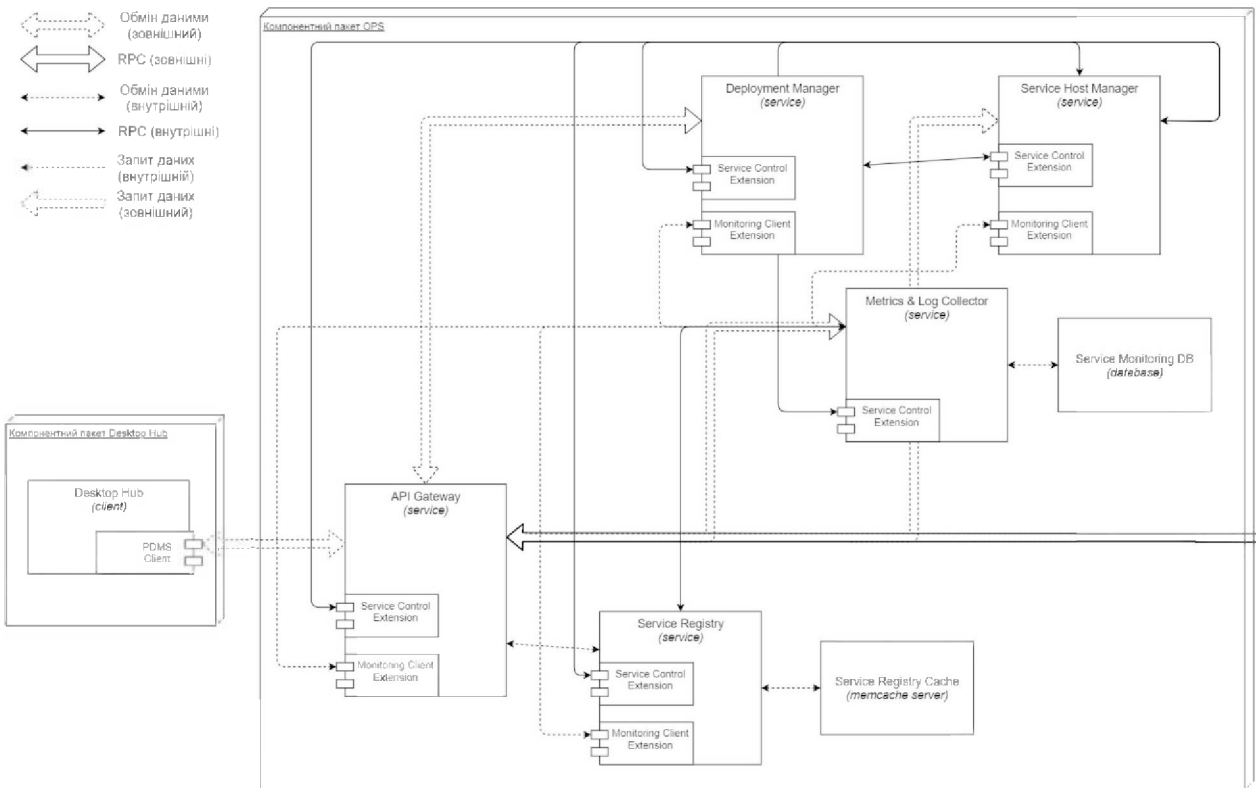

Рисунок 3.6 – схематичне відображення структурної схеми мікросервісного компоненту OPS

З використанням мікросервісного підходу, компонент є розподіленим на декілька окремих по функціональному принципу, тобто:

- API Gateway – є так званим “пропускним пунктом” для клієнтських запитів до цільових сервісів. Цей сервіс забезпечує виконання перевірок запитів та отримуваних даних, встановлених політик як щодо клієнту, так і щодо користувачів, що виконують запити, а також забезпечувати стабільну роботу системи та інформування клієнтів у разі проблем з роботою цільових сервісів, для чого він взаємодіє з сервісом Service Directory;
- Service Directory – є реєстром сервісів, що забезпечує контроль та взаємодію усіх інших сервісів у системі, підтримуючи динамічну натуру роботи, а також забезпечуючи інтеграцію без необхідності прямої залежності між сервісами. Цей сервіс не використовує постійного сховища даних через перемінність стану системи, у

тому числі активних сервісів, та відсутність необхідності збереження станів між перезапусками системи;

- Metrics & Log Container – виконує функції централізованого збору, обробки та збереження для подальшого зчитування та аналізу лог-даних та татистичних метрик роботи кожного сервісу у системі, а також відстеження статусу роботи сервісів для інформування реєстру та виконання подальших дій (перезапуск робочих одиниць сервісу, тощо);
- Deployment Manager – головний сервіс, що відповідає за управління розгортаннями сервісів, як основних компонентів, так і інтегрованих;
- Service Host Manager – є сервісом, що встановлюється на кожній хост-машині (так амо як інші сервіси – у контейнерному середовищі), є прошарком між загальним програмним кодом управління, що відноситься до комплексу ПЗ з однієї сторони, та технологічною і апаратною базою з іншої, дозволяючи виконувати операції з розгортання через запити до ПЗ оркестрації контейнерів, а також збирати статитичні дані з сервісів у сміжних контейнерах, та з апаратного забезпечення для контролю споживання ресурсів.

3.4.3 Робота з даними.

Робота з даними на сервісному рівні включає у себе проведення різних операцій, що є залежними від функціональності та вимог до кожного сервісу, але загалом, такі операції поділяються на наступні категорії:

- Обмін даними між клієнтом та сервером – може бути як у текстовому форматі (включаючи структуровані дані, у більшості випадків формату JSON), так і у двійковому, що зазвичай є інформацією для подальшого зберігання у спільному сховищі

(медіадані, структурована інформація інших пропрієтарних форматів, тощо). Особливістю сервісної архітектури є незмінність, тому зберігання даних у файлах на локальному сховищі системи де працює само ПЗ сервісу не є варіантом, особливо при роботі у контейнерах. За принципом, контейнери – тимчасові віртуальні простори, відокремлені від інших частин ОС, де не застосовується постійного сховища даних, а файлова система будується як з частин хост-системи, так і з образу збудованої системи, що може містити у собі й збудовані виконавчі файли сервісу.

За окремих випадків є можливість використання виділених точок прив'язки до файлової системи хосту, але це не є ідеальним методом збереження даних при розгортанні великої кількості одиниць сервісів, використання спільних сховищ, такі як БД, є більш оптимальним варіантом;

- Збереження або отримання даних з бази даних – у більшості випадків такі дані представляються у текстовому вигляді, структуровані згідно з структурою бази даних та таблиць, де окремим полям є відповідні поля значень у моделі даних;
- Міжсервісний обмін даними – як і у випадку обміну між клієнтами та сервісами, дані операції виконуються над даними, що цілком перебувають у пам'яті. Додатково, навідміну від операцій над даними у БД (у більшості випадків) для обміну між незалежними програмними ресурсами, особливо якщо це стосується великих об'ємів або важливих елементів інформації, є необхідним проведення перевірок цілісності та достовірності даних, а також, за необхідності – стиснення для економії пропускної здатності мережі взаємодії.

3.5 Огляд структури інструментальних (client) компонентів.

Структура клієнтського інструментального ПЗ орієнтована на забезпечення виконання завдань рівня представлення даних. Наявність окремих частин програмного коду слугує загалом для виконання службових функцій, або обробки локальної логіки та операцій, що не потребують додаткової взаємодії з сервісами.

3.5.1 Основні спільні технологічні та логічні частини.

Порівняно з сервісним ПЗ, архітектура клієнтських інструментальних додатків крім загальних спільних елементів з п. 3.1.4, є більш індивідуальною та залежить від необхідної функціональності та інтеграції рівнів представлення та обробки даних. Звичайним для таких випадків є застосування спільних існуючих бібліотек або лише архітектур для MVVM, а також інструментів відстеження та відправки інформації про помилки ПЗ.

Окремою ситуацією є клієнтське ПЗ для кінцевих продуктів. Через використання централізованих інструментів управління випуском та підтримкою продуктів постає питання забезпечення єдиної базової програмної архітектури для кожного програмного продукту. Вибором для вирішення такого завдання стала спільна програмна середа часу виконання, що об’єднує не лише технологічні елементи та базові логічні (описані у п. 3.1.4), а також надає інструменти для управління ПЗ програмних продуктів, включаючи наступні можливості:

- уніфікований процес інсталяції та видалення продуктів,
- єдина середа виконання для обсліговування декількох продуктів одночасно,
- функції автоматичного, автоматизованого та ручного оновлення (за налаштуваннями),

- підтримка концепцій віддаленої конфігурації та функціональних флагів, що дозволяють налаштовувати ПЗ у залежності від будь-яких параметрів, у тому числі і зовнішнього середовища (розташування коритувача, параметрів облікового запису, його статусу, та інших, залежно від продукта) у індивідуальному режимі;
- забезпечення збору та відправці даних телеметрії та лог-даних, завдяки єдиному механізму, що надається середою виконання,
- система модулів розширень, що надає продуктам можливість застосовувати окремі функціональні блоки, так звані “модулі розширень” з використанням єдиного API.

3.5.2 Основні відмінності.

Архітектура клієнтського ПЗ є кардинально відмінною від сервісної через декілька основних причин:

- Логічна частина та інтерфейс взаємодії займаються лише прийомом даних від сервісів, відображенням обробленої інформації користувачу, та передачею команд у іншу сторону на обробку до сервісів від самого користувача;
- Для окремих інструментів частина логіки також використовується і на клієнтській стороні. Такий код насамперед виконує більш допоміжні завдання, наприклад для обробки локальних файлів чи застосування ресурсів локального пристрою для відображення 3D-зображень;
- Оскільки авторитарним джерелом інформації є сервіси, то при обробці даних на клієнтській стороні у більшості випадків не є важливим валідація даних від користувача. Окремі випадки, де вона повинна застосовуватися – операції зі збереженням даних,

через необхідність підтримки цілісності та динамічність стану самої програми порівняно з файлами у сховищі або локальній БД.

3.5.3 Робота з даними.

Порівняно з операціями з даними, що виконуються на сервісній стороні, клієнтська частина комплексу ПЗ обмежена лише декількома варіантами:

- Зчитування та запис даних у локальне або спільне мережеве сховище (у бінарному або текстовому форматі).

  Операції щодо роботи з файлами доступні лише для клієнтів на мобільних платформах та ПК через обмеження веб-технологій, та здійснюються за рахунок API платформ, які надається базовими бібліотеками C#. Для спрощення використання цих можливостей, у набір спільних логічних елементів входить надбудова, що дозволяє у єдиному форматі працювати з файлами та директоріями на різних ОС, а також надає спільні методи для роботи як з стандартними операціями зчитування та запису, так і з файлами, що відображено у пам'яті. Такий механізм є особливо корисним для структур даних великого розміру, тому що у пам'ять передається лише окрема частина даних замість усього файлу.

  Для веб-додатків збереження файлів локально є обмеженим також і через максимальні кількість інформації у певних сховищах веб-браузера, виділених окремо для кожного сайту. Прицпово, це не є критичною проблемою через мету застосування та розроби таких додатків – відображення інформації, що потребує лише завантаження файлів з серверу, а реалізація цього механізму є стандартизованою та наявною як у стандартних API веб-браузерів, так і зі сторонніми бібліотеками для Javascript/TypeScript, які додають додаткові можливості по контролю за процесом запитів та їх параметрами, наприклад встановлення секретних ключів або

використанні облікових даних (з куків або іншим методом) для авторизації та доступу до персональної інформації;

- Передача та отримання даних від сервісів по протоколу HTTPS. Такий механізм взаємодії є важливим не лише для отримання файлів, наприклад з медіаданими, а й для обміну інформацією між спільними сервісами та індивідуальними одиницями клієнтського ПЗ, забезпечуючи як синхронізацію стану, так і виконання операцій на сервісній стороні з отриманням результату. Наявними API платформ, що надаються стандартними бібліотеками мов програмування доступні як окремі методи по відправці запитів (з різними типами, залежно від призначення запиту;наприклад, GET – для отримання інформації, POST – для запиту на виконання операції з відправленими даними), так і повноцінна реалізація різноманітних супутних методів у класі “веб-клієнт”. Такі методи взаємодії з сервісами є дуже важливими, наприклад, при необхідності забезпечення авторизації користувачів для доступу до функціоналу інструментного ПЗ, що реалізовано на прикладі компоненту Desktop Hub.

3.5.4 Реалізація інтерфейсу користувача – вибір native проти web.

Основними питаннями, на яких грунтується вибір базових технологій для розробки тої чи іншої частини ПЗ даного косплексу, є вибір між функціональністю, спеціалізацією, дизайном та сміжністю.

Зазвичай, вибір native, або розробки самостійного ПЗ, що працює на окремій платформі та використовує вбудовані можливості (ПК або мобільні), є при необхідності надання функціоналу додатку без веб-браузера або локально на окремому пристрої, без залучення сервісів. Так є для більшості програмної логіки, що входить у компонент Desktop Hub, він побудован саме як окремий додаток для ПК та використовує інтерфейсну платформу Avalonia для відображення графічних елементів. Перевагами такого вибору, окрім вище

зазначених, є швидкість реакції інтерфейсу на дії користувача, а також можливість роботи з локальними ресурсами, у тому числі з файлами у сховищі.

Для варіанту інтерфейсу native також є більш придатною архітектура MVVM, через більш точне розділення зон відповідальності, а також більшу самостійність окремого ПЗ, навідміну від веб-додатків.

Вибір проектування веб-додатку насамперед грунтується на необхідності створення так званого “фронтенду”, або клієнтської сторони сервісного компоненту, що може включати у себе як окремий функціонал, притаманний для можливостей сервісу або його частини, а також спільний з іншими веб-додатками, що дозволяє об’єдувати подібні за відношенням та функціональністю клієнти, додавати до них елементи, такі як заголовочний контейнер, що може включати у себе спільну навігацію між деклькома веб-додатками, спільний дизайн для естетичного об’єднання індивідуальних додатків у спільну систему, а також надавати доступ до загального для системи функціоналу, такому як управління акаунтом користувача. На рисунку 3.7 відображеий типовий вид такого контейнеру, що включає як спільну функціональність та елементи дизайну, так і індивідуальні для кожного додатку.

Рисунок 3.7 – типовий вигляд спільного заголовочного контейнеру для веб-додатків.

3.5.5 Використання спеціалізованих можливостей окремих програмних платформ.

Однією з особливостей мобільних платформ та проектування додатків для них – можливість виористання функціоналу так званих “push-повідомлень”, що дозволяють інформувати користувача про різні події прямо з відповідного сервісу, використовуючи лише API, що надаються

розробниками мобільних платформ, у даному випадку – Firebase Cloud Messaging для Android, та Apple Push Notification Service для iOS. Обидва засоби орієнтовані на індивідуальні платформи та використовують стандартний інтерфейс ОС для нформування користувача на екрані. Перевагами для виористання цих сервісів є вартість (безкоштовна), широкі ліміти навіть на безкоштовному плані, можливість відправки як стадартних шаблонних сповіщень, так і даних у своєму форматі, що після отримання будуть направлені до додатку для обробки [38, 39]. Однак, при застосуванні таких можливостей виникає недолік залежності від зовнішніх сервісів, що не є суміжним з вимогами до проекту. Не зважаючи на несуміжність, впровадження такої функції є корисним через можливість проактивного інформування як користувачів випущених програмних продуктів, так і для внутрішнього використання, інформуючи розробників про наступні типові події:

- вихід з ладу сервісу/апаратного забезпечення,
- критичні помилки у роботі компонентів,
- сигналізації щодо встановлених налаштувань як основних елементів комплексу ПЗ, так і інших інтеграцій (доступно через компонент OPS),
- інформування щодо необхідності підтвердження дій інших розробників, якщо у тих немає достатньо прав доступу до ресурсів чи операцій,
- інформування щодо спроб аутентифікації/авторизації у сервісах або інструментах системи, якщо налаштований 2FA.

У випадку з 2FA, використання окремого пристрою для підтвердження спроби входу в обліковий запис збільшує ефективність роботи та безпеку інформації, а для деяких пристроїв з вбудованим чипом NFC (Near Field Communication) є додаткова можливість застосування фізичної

персоналізованої картки розробника для підтвердження спроби, що ще більше підвищує надійність та безпеку системи.

У додатку Д наведено діаграми послідовності типового процесу аутентифікації користувача з налаштованою функцією 2FA. Два можливих випадки описують ситуації де у користувача є наявна активна авторизація у той самий обліковий запис у мобільному додатку, з використанням push-повідомлень для інформування про спробу входу, а інша – коли немає, з використанням унікально згенерованого QR-коду для отримання інформації щодо спроби входу.

Висновки до розділу 3.

При проектуванні комплексної системи з великою кількістю взаємно інтегрованого ПЗ є важливим визначення технічних, функціональних вимог, та створення загальної архітектурної моделі компонентів, що є складовими системи, а також визначення методів та інструментів забезпечення взаємодії між ними. Для конкретного проекту даної роботи було окремо проведено дослідження через поставлені вимоги щодо портативності, які торкнулися областей управління даними, відображення UI, обрання крос-платформних бібліотек, а також визначення переваг та необхідності вибору SQL-рішення для СУБД на сервісній частині компонентів.

ВИСНОВКИ

У цій кваліфікаційній роботі було проведено дослідження теоретичної проблематики систематизації та централізованого контролю та управління за процесами розробки та проектування програмних продуктів, огляд існуючих програмних архітектурних моделей, конкретних хмарних та само-розміщених рішень, виділено їх переваги та недоліки, та обґрунтуванню проведення експериментальної частини наукової роботи, а саме – проектування та розробка архітектурної моделі універсального комплексу інструментального та сервісного ПЗ для виконання завдань щодо надання розробникам та організаціям функціональних можливостей для автоматизації процесів, систематизації ведення проектної діяльності, а також централізованого контролю за ресурсами та доступом до них.

На основі спроектованої архітектури є наявною у процесі розробки програмна система, що буде виконувати додаткові завдання, пов’язані з специфікою роботи над модифікованими моделями даних щодо програмних продуктів проектів, з необхідністю підтримки їх роботи у різних так званих “всесвітах” (Universe), що забезпечує можливості по розділенню доступної інформації для існуючих груп користувачів, особливо щодо публічно доступних продуктів, та внутрішніх версій.

Загалом, використання обраної технологічної бази, а також принципів та підходів до проектування окремих компонентів, що базуються на публічно-доступних знаннях, включаючи шаблони проектування, дозволить досягнути поставленої мети та забезпечити впровадження необхідного функціоналу та його доступності на більшій кількості платформ та за меншим використанням ресурсів, ніж у існуючих рішеннь (як індивідуальних інструментах, та і повноцінних інтегрованих системах).

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Digital 2021 October Global Statshot Report / Datareportal. [Електронний ресурс]. режим доступу: URL: https://datareportal.com/global-digital-overview
2. Клієнт-серверна архітектура / Вікіпедія. [Електронний ресурс]. режим доступу: URL: https://uk.wikipedia.org/wiki/%D0%9A%D0%BB%D1%96%D1%94%D0%BD%D1%82-%D1%81%D0%B5%D1%80%D0%B2%D0%B5%D1%80%D0%BD%D0%B0_%D0%B0%D1%80%D1%85%D1%96%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0
3. What are microservices? [Електронний ресурс]. режим доступу: URL: https://microservices.io/
4. Microservice architecture pattern / Microservices.io. [Електронний ресурс]. режим доступу: URL: https://microservices.io/patterns/microservices.html
5. Microservices: Five Architectural Constraints / Nirmata. [Електронний ресурс]. режим доступу: URL: https://nirmata.com/2015/02/02/microservices-five-architectural-constraints/
6. The Philosophy of Microservice Architecture. [Електронний ресурс]. режим доступу: URL: https://web.archive.org/web/20161112001830/http://microservicesbook.io/the-philosophy-of-microservice-architecture/
7. Microservices adoption anti-patterns / Microservices.io. [Електронний ресурс]. режим доступу: URL: https://microservices.io/microservices/antipatterns/-/the/series/2019/06/18/microservices-adoption-antipatterns.html
8. Everything You Need To Know About Microservices Design Patterns / edureka. [Електронний ресурс]. режим доступу: URL: https://www.edureka.co/blog/microservices-design-patterns
9. DevOps / Вікіпедія. [Електронний ресурс]. режим доступу: URL: https://uk.wikipedia.org/wiki/DevOps
10. Лабенко Д.П., Більський Г.М. Наукова доповідь. «Модель програмного комплексу управління розробкою клієнт-сервісної системи». Матеріали семінару «Сучасні інформаційні технології» Харківського національного університету ім. В.Н. Каразіна. Харків, 2021 р.
11. Internet Traffic and Capacity in Covid-Adjusted Terms / Telegeography. [Електронний ресурс]. режим доступу: URL: https://blog.telegeography.com/internet-traffic-and-capacity-in-covid-adjusted-terms
12. GitLab Pricing / GitLab. [Електронний ресурс]. режим доступу: URL: https://about.gitlab.com/pricing/

13. Gitea vs GitLab / Gitlab. [Електронний ресурс]. режим доступу: URL: https://about.gitlab.com/devops-tools/gitea-vs-gitlab/
14. Gitea compared to other Git hosting options / Gitea Docs. [Електронний ресурс]. режим доступу: URL: https://docs.gitea.io/en-us/comparison/
15. Upgrade from an old Gitea / Gitea Docs. [Електронний ресурс]. режим доступу: URL: https://docs.gitea.io/en-us/upgrade-from-gitea/
16. Zero downtime upgrades / GitLab. [Електронний ресурс]. режим доступу: URL: https://docs.gitlab.com/ee/update/zero_downtime.html
17. SolarWinds hack explained: Everything you need to know / WhatIs. [Електронний ресурс]. режим доступу: URL: https://whatis.techtarget.com/feature/SolarWinds-hack-explained-Everything-you-need-to-know
18. Final: OpenID Connect Core 1.0 / OpenID. [Електронний ресурс]. режим доступу: URL: https://openid.net/specs/openid-connect-core-1_0-final.html
19. Access Control Models / UWHO Cyber Security. [Електронний ресурс]. режим доступу: URL: https://westoahu.hawaii.edu/cyber/best-practices/best-practices-weekly-summaries/access-control/
20. C++ performance vs. Java/C# / Stack Overflow. [Електронний ресурс]. режим доступу: URL: https://stackoverflow.com/questions/145110/c-performance-vs-java-c
21. Can a C# program be cross-platform? / Stack Overflow. [Електронний ресурс]. режим доступу: URL: https://stackoverflow.com/questions/6975207/can-a-c-sharp-program-be-cross-platform/47706150
22. The Roslyn .NET compiler provides C# and Visual Basic languages with rich code analysis APIs / GitHub. [Електронний ресурс]. режим доступу: URL: https://github.com/dotnet/roslyn
23. What is Xamarin? / Microsoft Docs. [Електронний ресурс]. режим доступу: URL: https://docs.microsoft.com/en-us/xamarin/get-started/what-is-xamarin
24. WPF Platform Setup – Xamarin / Microsoft Docs. [Електронний ресурс]. режим доступу: URL: https://docs.microsoft.com/en-us/xamarin/xamarin-forms/platform/other/wpf
25. GTK# Platform Setup – Xamarin / Microsoft Docs. [Електронний ресурс]. режим доступу: URL:: https://docs.microsoft.com/en-us/xamarin/xamarin-forms/platform/other/gtk?tabs=windows
26. When WPF will be cross-platform? / GitHub. [Електронний ресурс]. режим доступу: URL: https://github.com/dotnet/wpf/issues/3952
27. Components and Props / React. [Електронний ресурс]. режим доступу: URL: https://reactjs.org/docs/components-and-props.html
28. Why Should We Choose REST (Client-Server) Model to Develop Web Apps ? / Audira Zuraida. [Електронний ресурс]. режим доступу: URL: https://medium.com/@audira98/why-should-we-choose-rest-client-server-model-to-develop-web-apps-c3bb2451b13a

29. Introduction to gRPC / gRPC. [Електронний ресурс]. режим доступу: URL: https://grpc.io/docs/what-is-grpc/introduction/
30. The need for speed — Experimenting with message serialization / Hugo Vieira da Silva. [Електронний ресурс]. режим доступу: URL: https://medium.com/@hugovs/the-need-for-speed-experimenting-with-message-serialization-93d7562b16e4
31. SQL vs NoSQL / Guru99. [Електронний ресурс]. режим доступу: URL: https://www.guru99.com/sql-vs-nosql.html
32. Top 10 Databases to Use in 2021 / Towards Data Science. [Електронний ресурс]. режим доступу: URL: https://towardsdatascience.com/top-10-databases-to-use-in-2021-d7e6a85402ba
33. Graph Database vs Relational Database / Memgraph. [Електронний ресурс]. режим доступу: URL: https://memgraph.com/blog/graph-database-vs-relational-database
34. Server-side service discovery pattern / Microservices.io. [Електронний ресурс]. режим доступу: URL: https://microservices.io/patterns/server-side-discovery.html
35. Application metrics /Microservices.io. [Електронний ресурс]. режим доступу: URL: https://microservices.io/patterns/observability/application-metrics.html
36. Health Check / Microservices.io. [Електронний ресурс]. режим доступу: URL: https://microservices.io/patterns/observability/health-check-api.html
37. Distributed tracing / Microservices.io. [Електронний ресурс]. режим доступу: URL: https://microservices.io/patterns/observability/distributed-tracing.html
38. Firebase Cloud Messaging / Firebase Documentation. [Електронний ресурс]. режим доступу: URL: https://firebase.google.com/docs/cloud-messaging
39. User Notifications / Apple Developer Documentation. [Електронний ресурс]. режим доступу: URL: https://developer.apple.com/documentation/usernotifications