БАКАЛАВРСЬКА РОБОТА

БР. ІП - 04.00.00.000 ПЗ

Група ІП-22-4

Кузів Олег

2026

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Кузів Олег Михайлович

(прізвище, ім'я, по батькові)

УДК 004.4

(індекс)

БАКАЛАВРСЬКА РОБОТА

Розроблення платформи - фреймворку для створення хмарних застосунків

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

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

Здобувач освітнього рівня Кузів О.М.

(підпис, ініціали та прізвище здобувача)

Науковий керівник Піх Марія Михайлівна, асистент

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

Допущено до захисту
Завідувач кафедри

доц. Бандура В.В.

(посада) (підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2026

Івано-Франківський національний технічний університет нафти і газу

Інститут, факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доц. В.В. Бандура
“ ____ ” _____2026 р.

ЗАВДАННЯ

НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТОВІ

Кузіву Олегу Михайловичу

(прізвище, ім’я, по-батькові)

1. Тема проекту (роботи) “ Розроблення платформи - фреймворку для створення хмарних застосунків”

керівник проекту (роботи) Піх М.М., асистент

затверджені наказом закладу вищої освіти від **“** 21 ” квітня 2026р. № 179/7

2. Строк подання студентом проекту (роботи) 09 червня 2026 р.

3. Вихідні дані до проекту (роботи) Результати і матеріали отримані під час проходження переддипломної практики

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

1. Аналіз проблематики розробки хмарних застосунків та фреймворків
2. Методологічний базис та аналіз архітектурних рішень при проєктуванні застосунків
3. Проектування та інженерна реалізація фреймворку
4. Проектування, архітектура та реалізація фреймворку
5. Представлення інтерфейсу пропонованого фреймворку

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

1. Типові структури монолітних додатків (рис. 1.1, ст.14)
2. Приклад структури сервіс-орієнтованої архітектури (рис. 1.2, ст.15)
3. Архітектура Cloud Framework (рис. 1.3, ст. 16)
4. Архітектура нативних хмарних додатків (рис. 1.4, ст. 17)
5. Спрощена інфраструктура хмарного середовища (рис. 1.5, ст.20)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання **21 квітня 2026 р.**

Керівник _____
(підпис)

Завдання прийняв до виконання _____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту	Примітка
1	Аналіз проблематики розробки хмарних застосунків та фреймворків	04.05.2026	виконано
2	Методологічний базис та аналіз архітектурних рішень при проєктуванні застосунків	15.05.2026	виконано
3	Проектування та інженерна реалізація фреймворку	21.05.2026	виконано
4	Проектування, архітектура та реалізація фреймворку	28.05.2026	виконано
5	Представлення інтерфейсу пропонованого фреймворку	03.06.2026	виконано
6	Оформлення пояснювальної записки бакалаврської роботи завідувачем кафедри	09.06.2026	виконано

Студент – дипломник _____Кузів О.М.
(підпис)

Керівник роботи _____Піх М.М.
(підпис)

АНОТАЦІЯ

Бакалаврська робота містить 84 сторінки, 33 рисунки, список використаних джерел із 33 найменуваннями.

Метою роботи є розроблення платформи-фреймворку для створення хмарних застосунків, що забезпечує підвищення ефективності процесу розробки та автоматизацію ключових етапів життєвого циклу програмних застосунків.

Об'єкт дослідження - процеси проєктування та розробки хмарно-орієнтованих програмних систем.

Предмет дослідження - методи, моделі та засоби побудови фреймворків для створення хмарних застосунків, зокрема архітектурні рішення, механізми автоматизації та підходи до управління життєвим циклом програмних систем.

В першому розділі встановлено закономірності розвитку архітектурних підходів та обґрунтовано доцільність використання хмарно-орієнтованих технологій для побудови сучасних програмних систем

В другому розділі сформовано методологічний базис проєктування та визначено ключові атрибути якості хмарних застосунків

В третьому розділі розроблено та реалізовано концептуальну модель і архітектуру фреймворку Intelligent Cloud.

В четвертому розділі деталізовано архітектуру фреймворку та реалізовано його функціональні компоненти й інтерфейс

Висновок: розроблено ефективний платформний фреймворк для створення хмарних застосунків, що забезпечує автоматизацію розробки, масштабованість та підвищення якості програмних систем.

КЛЮЧОВІ СЛОВА: ХМАРНІ ОБЧИСЛЕННЯ, ФРЕЙМВОРК, ХМАРНІ ЗАСТОСУНКИ, МІКРОСЕРВІСНА АРХІТЕКТУРА, ORM, МЕТАДАНІ, МАСШТАБОВАНІСТЬ, АВТОМАТИЗАЦІЯ РОЗРОБКИ.

ANNOTATION

The bachelor's thesis contains 84 pages, 33 figures, a list of used sources with 33 names.

The method of work is the development of a platform-framework for creating cloud applications, which ensures increased efficiency of the development process and automation of key stages of the life cycle of software applications.

The object of research is the processes of design and development of cloud-oriented software systems.

The subject of research is methods, models and tools for building frameworks for creating cloud applications, in particular architectural solutions, automation mechanisms and approaches to managing the life cycle of software systems.

The first section establishes the regularity of the development of architectural approaches and justifies the expediency of using cloud-oriented technologies for building modern software systems.

The second section forms the methodological basis of design and defines the key attributes of the quality of cloud applications.

The third section develops and implements the conceptual model and architecture of the Intelligent Cloud framework.

The fourth section details the framework architecture and implements its functional components and interface.

Conclusion: an effective platform framework for creating cloud applications has been developed, which provides development automation, scalability and quality improvement of software systems.

KEYWORDS: CLOUD COMPUTING, FRAMEWORK, CLOUD APPLICATIONS, MICROSERVICE ARCHITECTURE, ORM, METADATA, SCALABILITY, DEVELOPMENT AUTOMATION.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

CRUD – Create, Read, Update, Delete

DDL – Data Definition Language

ER (ERD) – Entity-Relationship Diagram

IDE – Integrated Development Environment

JPA – Java Persistence API

JDBC – Java Database Connectivity

JWT – JSON Web Token

KPI – Key Performance Indicator

MDD (MDE) – Model-Driven Development (Engineering)

MVC – Model-View-Controller

ORM – Object-Relational Mapping

RAD – Rapid Application Development

TDD – Test-Driven Development

ВСТУП

Актуальність роботи

Сучасний етап розвитку інформаційних технологій характеризується стрімким переходом до хмарно-орієнтованих моделей обчислень, що забезпечують гнучкість, масштабованість та ефективне використання обчислювальних ресурсів. Зростання складності програмних систем, підвищення вимог до їх продуктивності, доступності та безпеки обумовлюють необхідність застосування нових підходів до їх проєктування та реалізації. У цьому контексті особливої актуальності набувають фреймворки як інструменти стандартизації процесів розробки, які дозволяють скоротити час створення програмного забезпечення, підвищити його якість та забезпечити повторне використання компонентів.

Розвиток мікросервісної архітектури, контейнеризації та оркестрації сервісів створює передумови для формування нових платформних рішень, орієнтованих на автоматизацію життєвого циклу програмних систем. Водночас існуючі фреймворки часто мають обмеження, пов’язані зі складністю інтеграції, недостатньою гнучкістю або високим порогом входження для розробників. Це зумовлює необхідність створення нових підходів до побудови фреймворків, які б поєднували у собі універсальність, масштабованість та простоту використання.

У даній роботі розглядається підхід до розроблення платформи-фреймворку для створення хмарних застосунків, що базується на використанні сучасних архітектурних принципів, методології метаданих та засобів автоматизації процесів розробки. Запропоноване рішення орієнтоване на підвищення ефективності створення програмних систем, зменшення складності їх підтримки та забезпечення високого рівня якості кінцевого продукту.

Актуальність теми дослідження зумовлена зростаючою роллю хмарних технологій у сучасній IT-індустрії, де вони виступають ключовим

інструментом побудови масштабованих, відмовостійких та гнучких інформаційних систем. Умови цифрової трансформації економіки вимагають від програмних рішень здатності швидко адаптуватися до змін бізнес-вимог, що неможливо без використання сучасних архітектурних підходів та інструментів автоматизації.

Незважаючи на значну кількість існуючих фреймворків для розробки хмарних застосунків, більшість із них орієнтовані на вирішення вузькоспеціалізованих задач або вимагають значних витрат часу на налаштування та інтеграцію. Це створює потребу у розробці універсальних платформних рішень, які забезпечують високий рівень абстракції та спрощують процес створення програмних систем.

Крім того, актуальність дослідження визначається необхідністю підвищення ефективності використання ресурсів, оптимізації процесів розробки та забезпечення високої якості програмного забезпечення. У зв'язку з цим розроблення платформи-фреймворку для створення хмарних застосунків є важливим науково-практичним завданням, що має значний потенціал для впровадження в реальних умовах.

Метою роботи є розроблення платформи-фреймворку для створення хмарних застосунків, що забезпечує підвищення ефективності процесу розробки та автоматизацію ключових етапів життєвого циклу програмних застосунків.

Об'єкт дослідження - процеси проєктування та розробки хмарно-орієнтованих програмних систем.

Предмет дослідження - методи, моделі та засоби побудови фреймворків для створення хмарних застосунків, зокрема архітектурні рішення, механізми автоматизації та підходи до управління життєвим циклом програмних систем.

Завдання роботи:

1. Провести аналіз еволюції архітектурних парадигм програмних систем.
2. Дослідити теоретичні засади хмарних обчислень та їх сучасні моделі.

3. Проаналізувати методологічні підходи до проєктування програмних систем.

4. Визначити ключові атрибути якості хмарно-орієнтованих застосунків.

5. Розробити концептуальну модель фреймворку для створення хмарних застосунків.

6. Реалізувати архітектуру та основні компоненти фреймворку.

Методи дослідження

У роботі використано комплекс методів наукового дослідження, зокрема методи системного аналізу для дослідження архітектурних підходів, методи порівняльного аналізу для оцінки існуючих рішень, методи моделювання для побудови архітектури фреймворку, а також методи об’єктно-орієнтованого та компонентного проєктування для реалізації програмного забезпечення.

Наукова новизна

Наукова новизна роботи полягає у розробленні комплексного підходу до створення платформи-фреймворку для хмарних застосунків, який поєднує методологію проєктування на основі метаданих, принципи мікросервісної архітектури та механізми автоматизованої генерації програмних компонентів.

Практичне застосування

Практична цінність отриманих результатів полягає у створенні функціонального фреймворку, який може бути використаний для розробки хмарних застосунків у різних предметних областях, забезпечуючи скорочення часу розробки, підвищення якості програмного забезпечення та зниження витрат на його супровід.

Бакалаврська робота містить 84 сторінки, 33 рисунки, 4 розділи, переліки використаних джерел налічує 33 найменування.

РОЗДІЛ 1. АНАЛІЗ ПРОБЛЕМАТИКИ РОЗРОБКИ ХМАРНИХ ЗАСТОСУНКІВ ТА ФРЕЙМВОРКІВ

1.1. Еволюція архітектурних парадигм та передумови розробки хмарно-орієнтованих застосунків

Сучасна цифрова трансформація зумовлює впровадження програмних рішень критичного масштабу в усіх сферах життєдіяльності. Суб'єкти державного сектору реалізують системи електронного урядування (e-government) для надання сервісів мільйонам громадян, що потребує високого рівня інтеграції внутрішніх адміністративних процесів. У науковому середовищі спостерігається тенденція до створення розподілених систем управління великими даними з розвиненими інструментами візуалізації для міждисциплінарної взаємодії.

1.1.1. Аналіз монолітної архітектури та обмеження її застосування

Історично домінуючим підходом у програмній інженерії була монолітна архітектура, що передбачає побудову додатка як єдиного нероздільного дистрибутивного блоку. Дана парадигма була ефективною в умовах обмеженої функціональної складності, стабільності вимог та відносно невеликої аудиторії користувачів. Основні характеристики та приклади типових структур монолітних додатків наведено на рисунку 1.1.

Проте розвиток екосистеми хмарних обчислень, Інтернету речей (IoT) та зростання вимог до швидкості виходу продукту на ринок (Time-to-Market) виявили суттєві недоліки монолітного підходу:

- Складність масштабування: необхідність реплікації всього додатка замість його окремих високонавантажених компонентів.

- Технологічна ригідність: неможливість використання різних стеків технологій у межах одного проєкту.

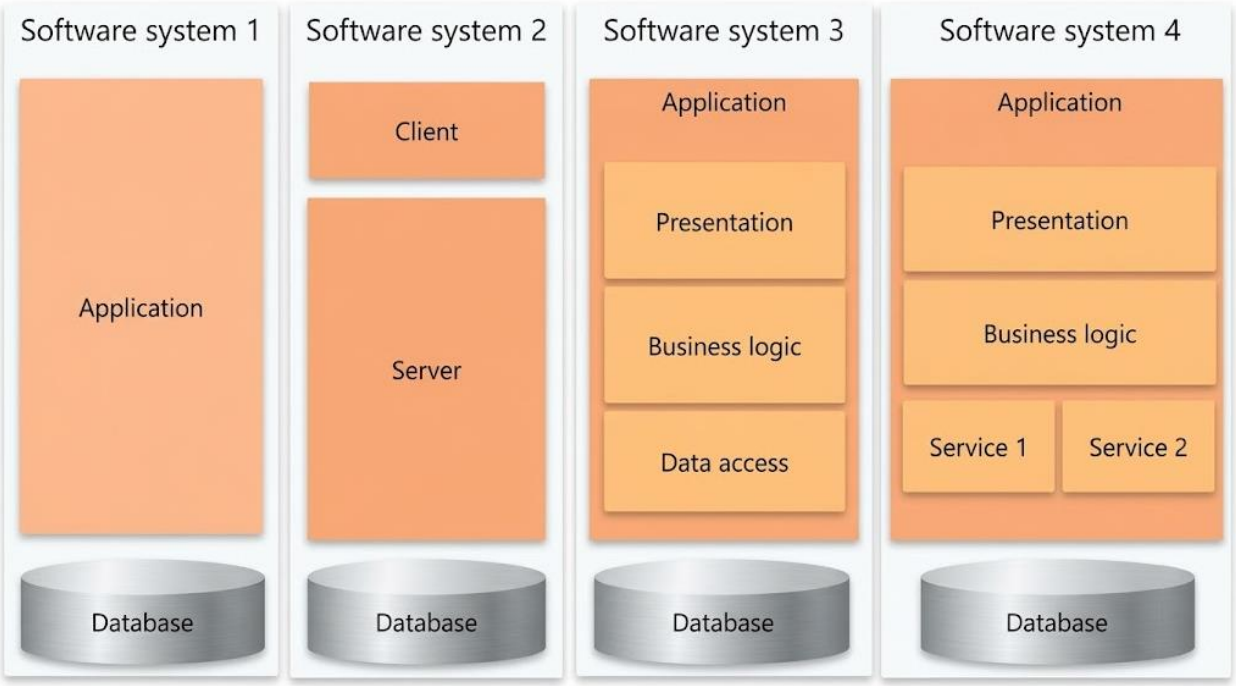


Рисунок 1.1 - Типові структури монолітних додатків

- Ризики розгортання: мінімальна зміна в коді потребує повного перескладання та тестування всієї системи, що збільшує імовірність критичних помилок.

1.1.2. Сервіс-орієнтована та мікросервісна архітектури

Як альтернатива моноліту була запропонована сервіс-орієнтована архітектура (SOA), що базується на принципах інкапсуляції бізнес-логіки в самодостатні сервіси, які взаємодіють через стандартні протоколи, переважно SOAP. Попри концептуальну прогресивність, SOA не отримала повсюдного поширення в динамічних проєктах через надмірну складність комунікаційних протоколів («важковаговість») та жорстку прив'язку до централізованих реляційних баз даних, що створювало інфраструктурні обмеження.

На рисунку 1.2 подано основні компоненти такої архітектури:

- клієнти: веб-додаток, мобільний додаток та інші системи.

- ESB (Enterprise Service Bus): центральна шина, яка оркеструє взаємодію.

- реєстр сервісів: для їх виявлення та публікації.

- сервіси: наприклад, сервіс користувачів, сервіс замовлень, сервіс інвентаризації, кожен з яких є автономним модулем.

- спільна база даних: у нижній частині діаграми, що показує спільний доступ до даних між сервісами,

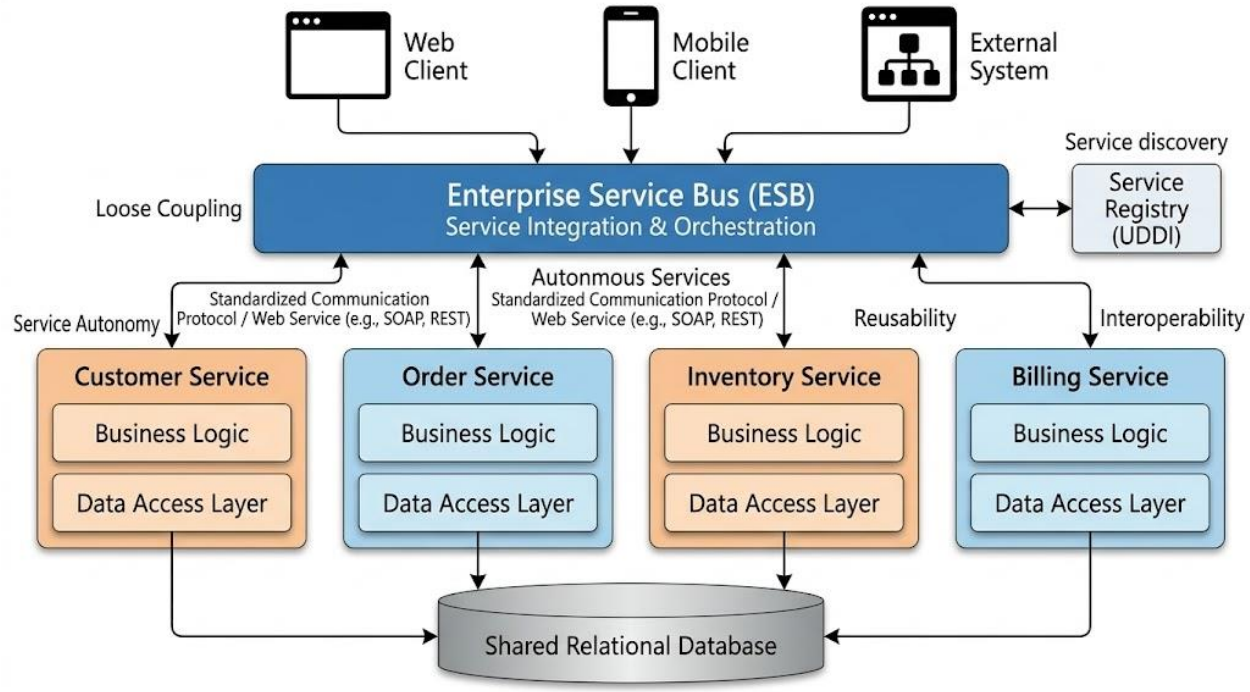


Рисунок 1.2 – Приклад структури сервіс-орієнтованої архітектури

Еволюційним кроком стало формування мікросервісного архітектурного стилю. В межах MsA програмний продукт розглядається як сукупність автономних, легкоатлантних компонентів, що мають власні механізми збереження даних (Persistence) та взаємодіють за допомогою протоколів з низькими накладними витратами (наприклад, REST поверх HTTP).

Ключові переваги MsA:

- Незалежність розгортання: кожен сервіс може бути оновлений без зупинки всієї системи.

- технологічна поліморфність: можливість поєднання різних мов програмування та фреймворків (наприклад, Java Spring, Python, Node.js) залежно від специфіки задачі.

- стійкість до відмов: ізоляція помилок у межах одного мікросервісу.

1.1.3. Хмарно-орієнтовані додатки

Найбільшу ефективність MsA демонструє в середовищах Cloud Native. Хмарно-орієнтовані додатки (CNA) проектуються спеціально для експлуатації в моделях IaaS/PaaS, забезпечуючи високу доступність, еластичність та автоматизований моніторинг. Проте перехід до CNA пов'язаний із низкою бар'єрів: високим порогом входження, дефіцитом експертних знань та складністю забезпечення переносності (portability) між різними хмарними провайдерами.

Враховуючи окреслені виклики, актуальним постає питання розробки методології, яка б забезпечила типізацію та прискорення процесу створення CNA. У даній роботі пропонується концептуальний підхід Smart-Cloud, що інтегрує принципи багаторазового використання компонентів та ліній програмних продуктів (Software Product Lines). Загальна високорівнева архітектура підходу представлена на рисунку 1.3.

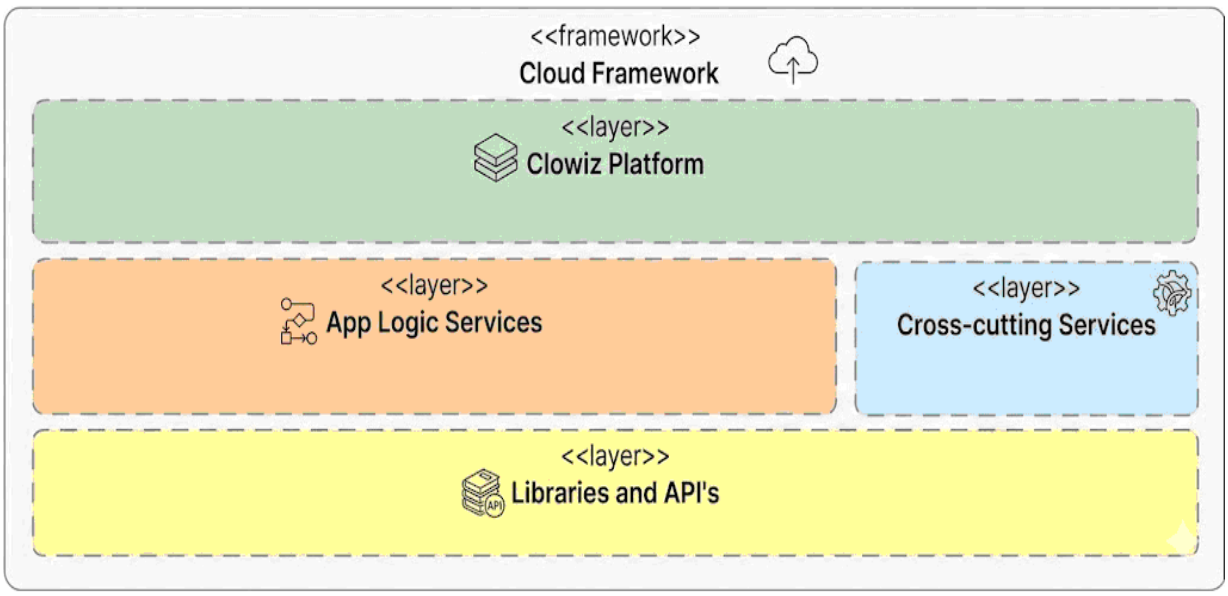


Рисунок 1.3 - Архітектура Cloud Framework

Додатково розроблено референтну архітектуру нативних хмарних додатків, яка структурує взаємодію мікросервісів у хмарному середовищі (рисунок 1.4).

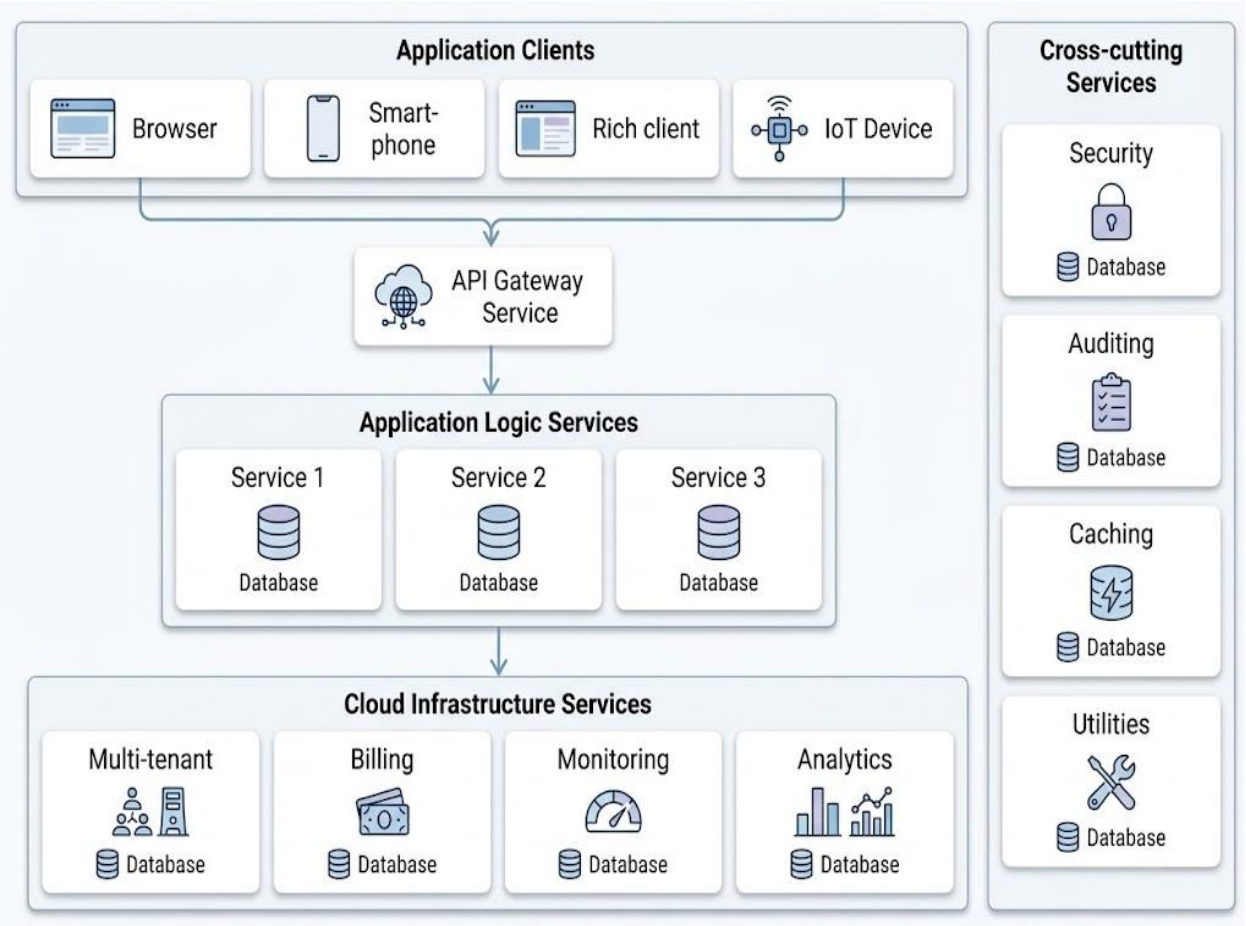


Рисунок 1.4 - Архітектура нативних хмарних додатків

Практична реалізація дослідження ґрунтується на використанні уніфікованої мови моделювання (UML) та сучасних інженерних практик:

- технологічний стек: фреймворки на базі Java, що забезпечують модульність.

- методи розробки: впровадження конвеєрів безперервної інтеграції та доставки (CI/CD) за допомогою інструментів автоматизації тестування та розгортання.

Валідація запропонованих рішень здійснювалася шляхом розробки платформи для візуального моделювання хмарних інформаційних систем без

безпосереднього написання коду (Low-code/No-code approach).

1.2. Теоретичні засади та генезис парадигми хмарних обчислень

1.2.1. Концептуалізація хмарних технологій у сучасній IT-інфраструктурі

Трансформація галузі інформаційних технологій під впливом хмарних обчислень зумовила фундаментальний перехід від володіння фізичними активами до моделі споживання ресурсів як послуги. Економічна доцільність хмарної парадигми базується на принципі високої щільності використання ресурсів: оренда значних обчислювальних потужностей (наприклад, 1000 віртуальних вузлів) на короткий термін є фінансово вигіднішою та ефективнішою, ніж тривале утримання обмеженої кількості власних серверів. Для сегмента стартапів та малого бізнесу хмарні технології стали каталізатором зростання, забезпечивши доступ до інфраструктури корпоративного рівня без значних капітальних інвестицій.

Динаміка ринку підтверджує цей тренд: за даними аналітичного агентства Gartner, світові витрати на публічні хмарні сервіси, що становили близько $210 млрд у 2016 році, продемонстрували стабільне зростання з прогнозованим досягненням позначки у $383 млрд у 2022 році.

Хмарні обчислення нівелюють необхідність у розгортанні локальних дата-центрів, дозволяючи розробникам зосередитися на створенні та миттєвому впровадженні вебдодатків. Великі корпорації трансформують свої стратегії, відмовляючись від закупівлі складного пропрієтарного ПЗ, яке швидко старіє та потребує високих операційних витрат на підтримку. В умовах домінування мобільного Інтернету, який за обсягами трафіку та кількістю підключених пристроїв суттєво випереджає традиційний десктопний сегмент, хмарна архітектура стає єдиним життєздатним середовищем для забезпечення масштабованості.

Впровадження хмарних обчислень актуалізувало низку технологічних напрямів:

- Мобільні інтерактивні системи: забезпечення низької затримки для кінцевих користувачів.

- Паралелізація обчислень: можливість одночасного виконання масивних обчислювальних завдань.

- Управління великими даними (Big Data): обробка експоненціально зростаючих обсягів інформації.

- Інтернет речей (IoT) та граничні обчислення (Edge Computing): необхідність аналітики в реальному часі та зберігання даних поблизу джерела їх генерації.

Під хмарними обчисленнями розуміється великомасштабна розподілена парадигма, що базується на економії масштабу та надає пул абстрагованих, віртуалізованих і динамічно масштабованих обчислювальних ресурсів (потужностей, сховищ, платформ) через мережу Інтернет на вимогу зовнішніх контрагентів.

Відповідно до дефініції NIST, хмарна модель включає п’ять атрибутів:

- Самообслуговування на вимогу (On-demand self-service): автоматизоване отримання ресурсів користувачем без прямої взаємодії з персоналом постачальника.

- Широкий мережевий доступ (Broad network access): доступність сервісів через стандартні протоколи для різноманітних клієнтських платформ.

- Об’єднання ресурсів (Resource pooling): використання багатокористувацької моделі (multi-tenancy) для динамічного розподілу фізичних та віртуальних потужностей між багатьма споживачами.

- Миттєва еластичність (Rapid elasticity): здатність системи до оперативного масштабування в обох напрямках залежно від поточного навантаження.

- Вимірювана послуга (Measured service): автоматичний моніторинг та білінг ресурсів на основі фактичного споживання.

Концептуальна схема спрощеної хмарної інфраструктури представлена на рисунку 1.5.

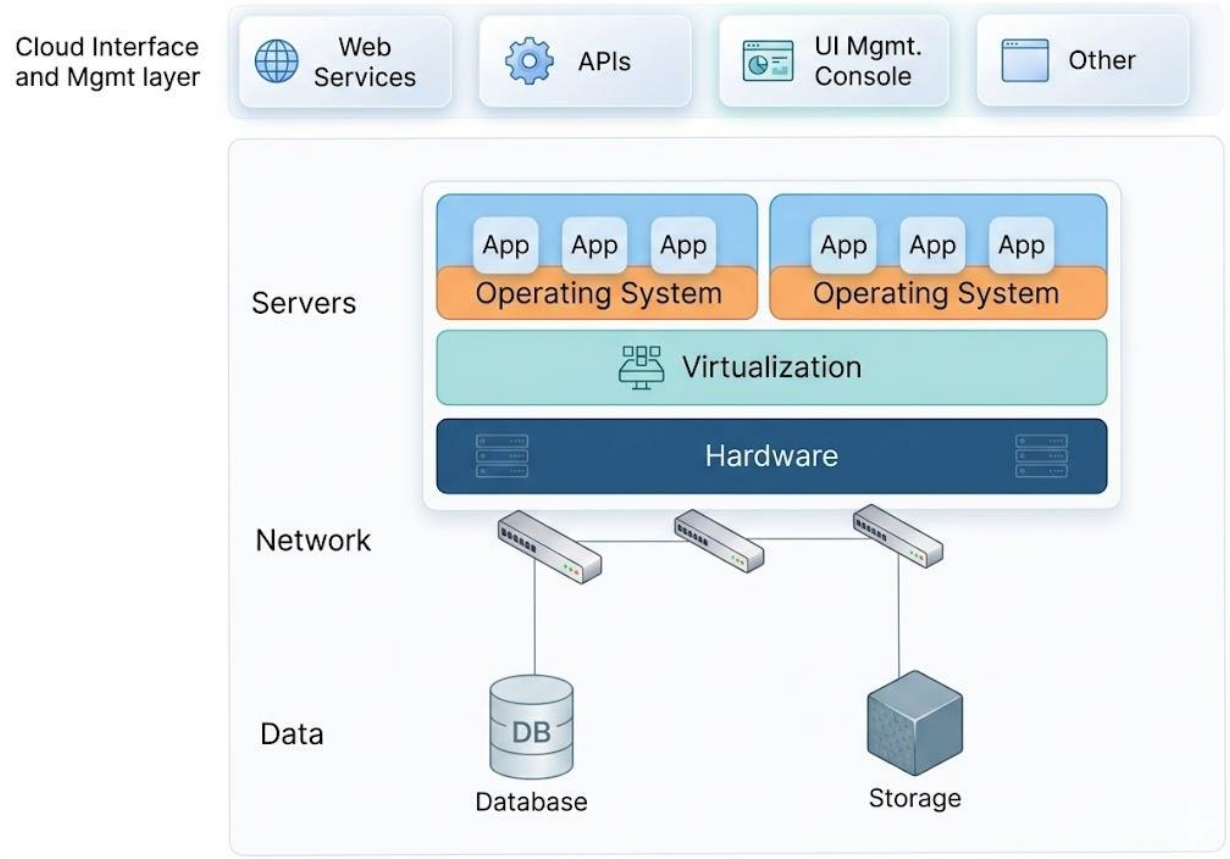


Рисунок 1.5 - Спрощена інфраструктура хмарного середовища

1.2.2. Ретроспективний аналіз та еволюція обчислювальних моделей

Ідейне коріння хмарних обчислень сягає середини XX століття, базуючись на концепції комунальних обчислень (Utility Computing). Ця модель аналогічна споживанню електроенергії чи водопостачання: клієнт сплачує лише за фактично використаний обсяг послуги, що стає можливим завдяки спільному використанню магістральної інфраструктури багатьма абонентами, що суттєво знижує питому вартість ресурсу.

Розглянемо історичні етапи формування парадигми

Етап зародження ідей (1960-ті рр.): Джон Маккарті у 1961 році вперше висунув гіпотезу про надання обчислювальної потужності як публічної послуги. Потім було деталізовано деталізував переваги розподілу ресурсів,

вказавши на можливість усунення локальних сховищ та стимулювання розвитку розподілених систем, попри тогочасні бар'єри у вигляді дорогого апаратного забезпечення.

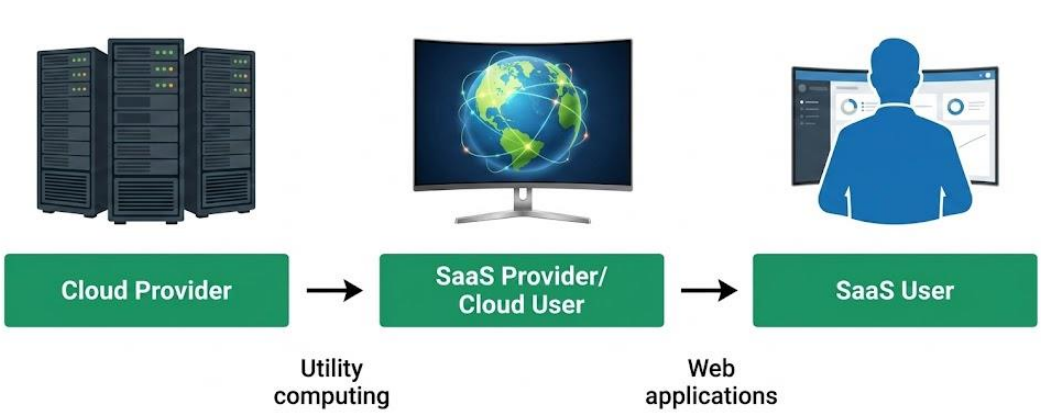


Рисунок 1.6 – Постачальники та користувачі хмарних сервісів

Технологія розподілу часу (Time-sharing). Компанії (зокрема IBM) впроваджували механізми спільного використання мейнфреймів декількома користувачами (рисунок 1.7). Проте через закритість архітектур та високу вартість ліній зв'язку ці рішення залишалися локальними.

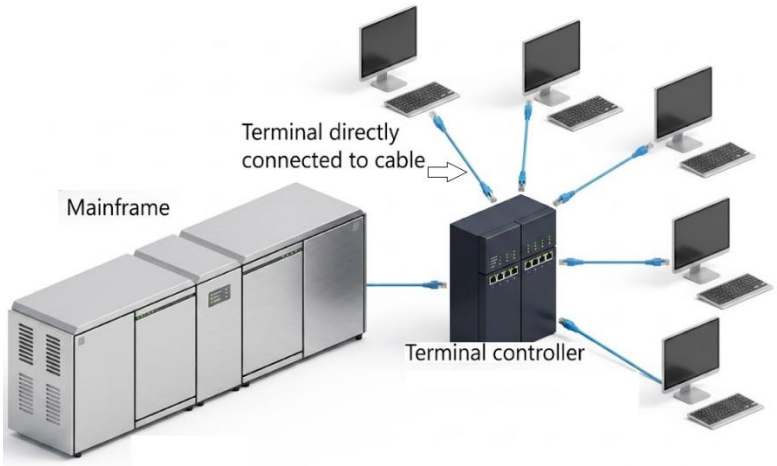


Рисунок 1.7 - Модель розподілу часу на базі мейнфреймів

Епоха Інтернету та стандартизації (1980–1990-ті рр.) - поява персональних комп'ютерів та розробка стеків протоколів TCP/IP, HTTP, SSL створили фундамент для глобальної мережевої взаємодії.

Сітчасті обчислення (Grid Computing) (початок 2000-х рр.) - модель Grid дозволила об'єднувати географічно розподілені вузли для виконання складних завдань, що безпосередньо еволюціонувало в хмарні системи з появою досконалих технологій віртуалізації.

На перехід до хмарної моделі вплинули як технічні, так і економічні детермінанти. У Microsoft Research у 2003 році підкреслено критичну важливість автоматизації: приклад Google, де 25 операторів керували 10 000 серверами, продемонстрував шлях до радикального зниження операційних витрат.

Технічні чинники:

- Зниження вартості компонентів та розвиток Open Source ПЗ.

- Повсюдне поширення смартфонів та бездротового зв’язку, що сформувало постійний попит на хмарні бекенд-сервіси.

- Зрілість технологій віртуалізації, що забезпечили ізоляцію та ефективне мультитенентне використання апаратних ресурсів.

Отже, сучасні хмарні обчислення є результатом конвергенції десятиліть технологічного розвитку, що трансформували обчислення з дефіцитного продукту в загальнодоступний комунальний ресурс.

1.3. Таксономія моделей розгортання та сервісної ієрархії хмарних обчислень

Ефективність впровадження хмарних технологій значною мірою залежить від вибору адекватної моделі розгортання, що визначається цільовою аудиторією, вимогами до безпеки та специфікою керування ресурсами.

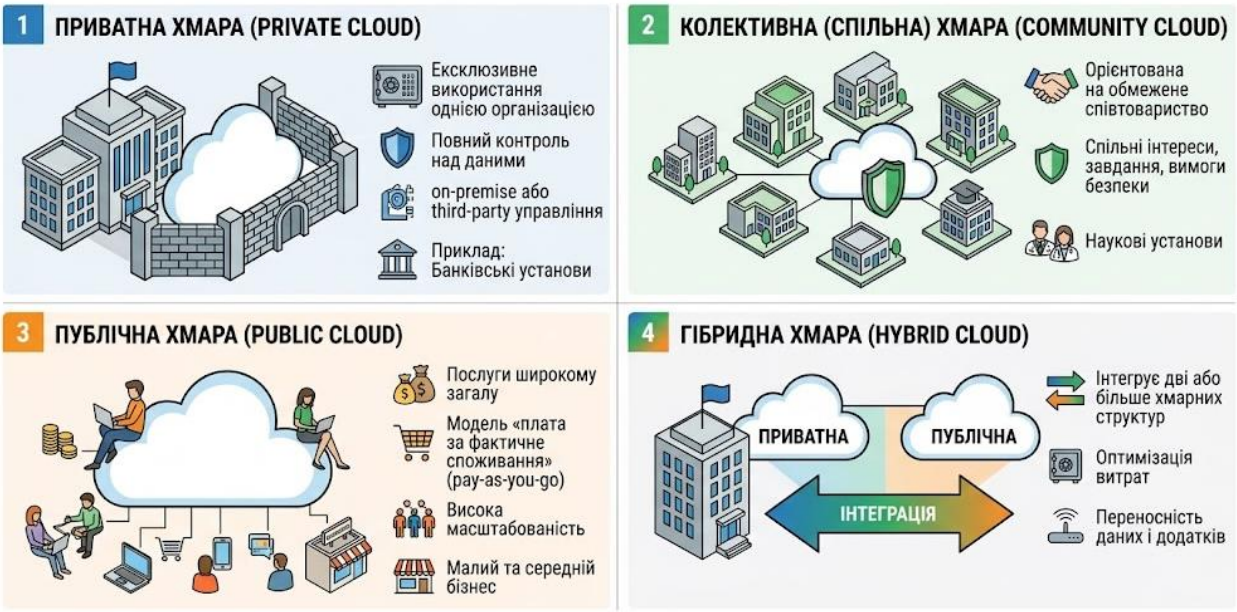


Рисунок 1.8 - Моделі розгортання хмарних інфраструктур

Згідно з усталеною класифікацією, виділяють чотири основні моделі розгортання хмарних інфраструктур (рис. 1.8):

- Приватна хмара (Private Cloud): являє собою інфраструктуру, призначену для ексклюзивного використання однією організацією, що може охоплювати декілька споживчих підрозділів. Вона може функціонувати на базі власних потужностей організації (on-premise) або керуватися третьою стороною. Ключовою перевагою є повний контроль над даними та відповідність специфічним регуляторним вимогам, що є критичним, наприклад, для банківських установ при розгортанні основних операційних систем.

- Колективна (спільна) хмара (Community Cloud): орієнтована на обмежене співтовариство користувачів із спільними інтересами, завданнями чи вимогами до безпеки.

- Публічна хмара (Public Cloud): передбачає надання послуг широкому загалу на основі моделі «плата за фактичне споживання» (pay-as-you-go). Ця модель характеризується високим рівнем масштабованості та є найбільш затребуваною серед малого та середнього бізнесу завдяки відсутності капітальних витрат на інфраструктуру.

- Гібридна хмара (Hybrid Cloud): інтегрує дві або більше хмарних структур (приватних, колективних чи публічних), які зберігають свою унікальність, але об'єднані стандартизованими технологіями, що забезпечують переносність даних і додатків. Це дозволяє організаціям оптимізувати витрати, використовуючи публічні ресурси для некритичних навантажень та приватні — для конфіденційних даних (рисунок 1.9).

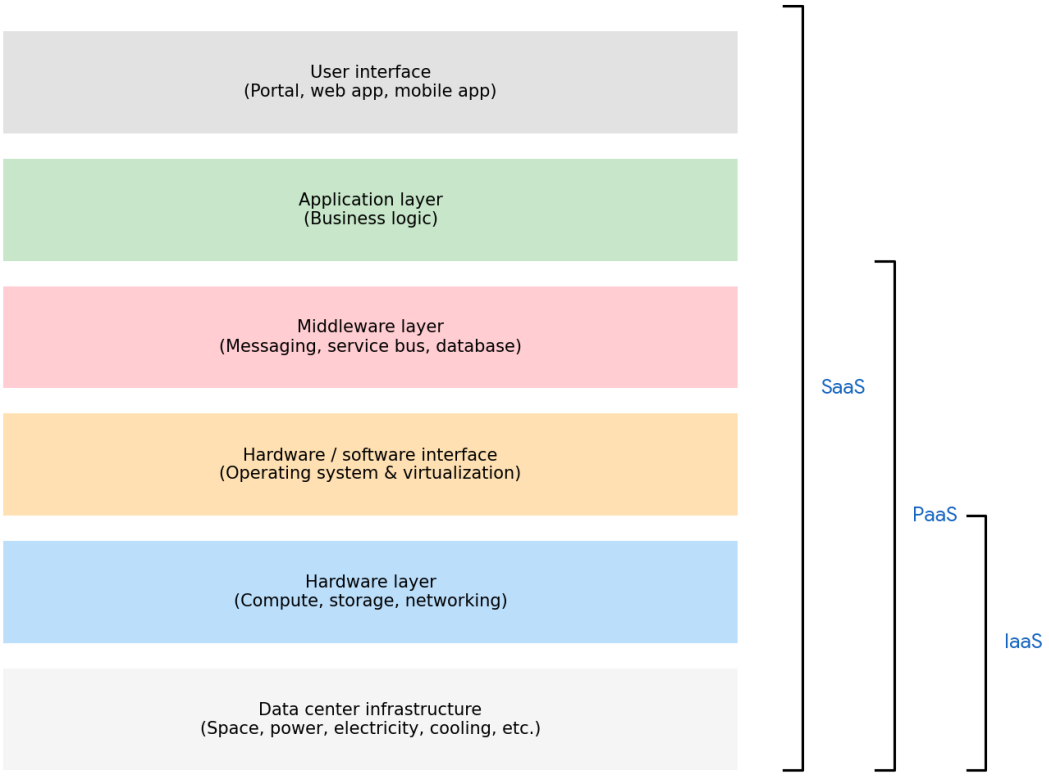


Рисунок 1.9 – Моделі хмарних послуг

Архітектура хмарних обчислень базується на ієрархічній структурі надання послуг. Відповідно до стандартів NIST, виділяють три фундаментальні моделі:

IaaS є базовим рівнем, де постачальник надає фундаментальні обчислювальні ресурси: віртуальні сервери, сховища даних, мережеві компоненти та засоби балансування навантаження. Споживач отримує можливість розгортати довільне програмне забезпечення, включаючи

операційні системи, проте не має прямого контролю над базовим фізичним обладнанням хмари.

PaaS функціонує над рівнем IaaS і надає середовище для розробки, тестування та розгортання програмних продуктів. Користувач використовує мови програмування, бібліотеки та інструменти розробки, підтримувані постачальником, не піклуючись про налаштування серверів чи баз даних. Це значно прискорює цикл розробки, оскільки керування базовою інфраструктурою повністю делеговано провайдеру.

SaaS є найвищим рівнем абстракції, де кінцевий користувач отримує доступ до готових прикладних додатків через Інтернет. Згідно з архітектурними принципами, наприклад, розробленими в IBM, SaaS має базуватися на мультитенантній моделі, де єдиний екземпляр системи обслуговує багатьох клієнтів. Користувачі не мають впливу на код чи інфраструктуру, але можуть здійснювати конфігураційні налаштування під власні потреби. Типовими прикладами є системи електронної пошти, CRM-рішення та віртуальні робочі столи.

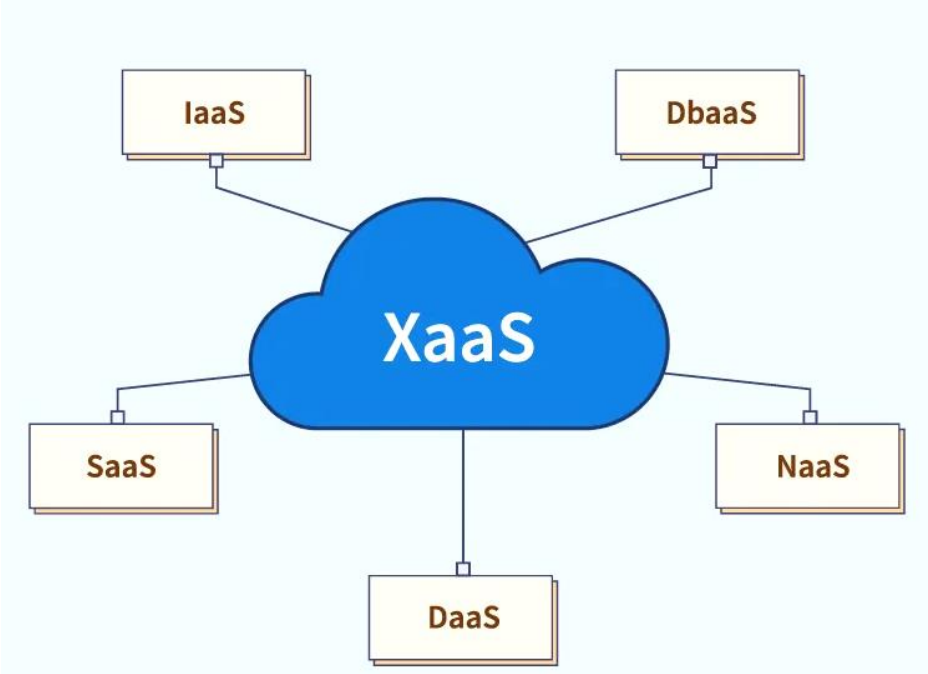


Рисунок 1.10 - Концепція XaaS

Концепція XaaS (Anything as a Service) (рисунок 1.10). Розвиток хмарних технологій призвів до появи спеціалізованих сервісних моделей, що охоплюють вузькі галузі:

- Контейнер як послуга (CaaS) — управління контейнеризованими додатками.

- База даних як послуга (DBaaS) — надання масштабованих систем керування базами даних.

- Безпека як послуга (SecaaS) — делегування функцій кіберзахисту зовнішньому провайдеру.

- Тестування як послуга (TaaS) — використання хмарних потужностей для верифікації ПЗ.

Усі вищезазначені варіації інтегруються в уніфіковану парадигму XaaS, де будь-який IT-актив може бути наданий як сервіс, що забезпечує максимальну гнучкість сучасної цифрової екосистеми.

1.4. Висновки по розділу

У результаті проведеного аналізу встановлено, що еволюція архітектурних парадигм програмних систем є закономірним процесом, зумовленим зростанням вимог до масштабованості, гнучкості та ефективності обробки даних. Дослідження монолітної архітектури дозволило виявити її ключові обмеження, зокрема складність супроводу, низьку адаптивність до змін та обмежені можливості горизонтального масштабування.

Обґрунтовано доцільність переходу до сервіс-орієнтованих і мікросервісних підходів, які забезпечують декомпозицію системи та підвищення рівня незалежності її компонентів.

Проаналізовано теоретичні засади хмарних обчислень, що дало змогу визначити їх роль як основи сучасної IT-інфраструктури та платформи для

побудови розподілених систем. Систематизація моделей розгортання та сервісних моделей хмарних обчислень створила концептуальне підґрунтя для подальшого проєктування хмарно-орієнтованих фреймворків.

РОЗДІЛ 2. МЕТОДОЛОГІЧНИЙ БАЗИС ТА АНАЛІЗ АРХІТЕКТУРНИХ РІШЕНЬ ПРИ ПРОЄКТУВАННІ ПРОГРАМНИХ ЗАСТОСУНКІВ

2.1. Теоретико-методологічні засади архітектурного проєктування програмних систем

Архітектура програмного забезпечення виступає фундаментальним інструментом нівелювання розриву між специфікаціями вимог та програмною реалізацією кінцевого продукту. У контексті розробки складних систем середнього та великого масштабу архітектурні рішення набувають критичного значення, оскільки вони безпосередньо детермінують успішність реалізації проєкту. Виважений архітектурний підхід забезпечує стабільність та життєздатність системи, тоді як ігнорування етапу проєктування або прийняття хибних рішень може призвести до технологічної деградації продукту. За своєю суттю, архітектура програмного забезпечення — це сукупність структурованих абстракцій, що включають програмні елементи, існуючі між ними відношення та їхні специфічні властивості. Вона є базисом для концептуального аналізу системи та досягнення необхідних атрибутів якості.

Атрибути якості (традиційно відомі як нефункціональні вимоги) визначають рівень відповідності системи потребам користувачів та бізнесу. Більшість із них є емергентними властивостями, тобто такими, що проявляються повною мірою лише після розгортання ПЗ в реальному експлуатаційному середовищі.

Процес архітектурної декомпозиції системи на окремі компоненти є ключовим для забезпечення таких показників:

- Модифікованість (Modifiability) — здатність системи до адаптації та внесення змін без значних витрат ресурсів.

- Доступність (Availability) — гарантування працездатності системи протягом визначеного часу.

- Надійність (Reliability) — здатність ПЗ безвідмовно виконувати функції за заданих умов.

- Масштабованість (Scalability) — можливість системи нарощувати продуктивність пропорційно до збільшення навантаження.

- Безпека (Security) — захищеність даних та функціональних можливостей від несанкціонованого доступу.

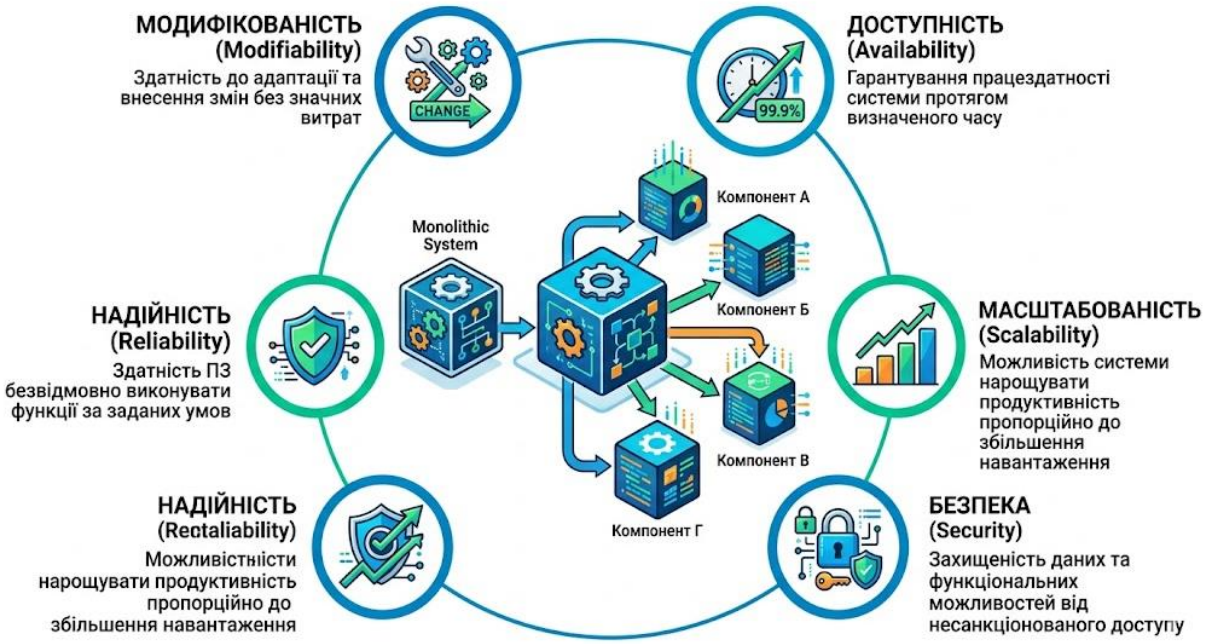


Рисунок 2.1 – Візуалізація архітектурної декомпозиції

Архітектура також виконує роль обмежувального фактора, визначаючи допустимі межі майбутніх модифікацій системи.

2.1.1. Функціональне призначення архітектурних рішень

Впровадження архітектурного підходу в життєвий цикл розробки ПЗ реалізує наступні завдання:

- Архітектурне проєктування: формування вхідних даних для етапу будівництва системи.

- Документування та візуалізація: створення графічних схем та діаграм для ефективної комунікації зі стейкхолдерами (зацікавленими сторонами).

- Управління знаннями: трансляція технічного досвіду та навчання нових учасників команди.

- Раціоналізація рішень: надання обґрунтування обраних методів реалізації та патернів проєктування.

- Верифікація та оцінка: аналіз якості існуючих систем та прогнозування їхньої поведінки.

Процеси формування архітектури можна систематизувати за рівнем їхньої формалізації:

- Неформальний: характеризується мінімальним документуванням та використанням вільних технік візуалізації (ескізи, неструктуровані діаграми) для опису найбільш критичних компонентів.

- Напівформальний: передбачає часткове використання галузевих стандартів моделювання.

- Формальний: базується на суворому дотриманні стандартів (наприклад, UML), повній документації та регламентованих процедурах перевірки.

2.1.2. Переваги та стратегічні виклики

Належне впровадження архітектурних практик забезпечує глибоке розуміння системних процесів, стимулює повторне використання компонентів та підвищує загальну якість будівництва ПЗ. Це веде до оптимізації процесів супроводу та розвитку програмного продукту.

Попри очевидні переваги, архітектурне проєктування стикається з низкою бар'єрів:

- Дефіцит кваліфікованих кадрів: брак фахівців з архітектурним мисленням та глибокими експертними знаннями.

- Економічні чинники: необхідність значних початкових інвестицій у проєктування на ранніх етапах проєкту.

- Управлінські ризики: недооцінка менеджментом довгострокових переваг архітектури на користь короткострокових результатів розробки.

2.2. Методологія лінійок програмних продуктів як парадигма системного повторного використання

У сучасній програмній інженерії концепція лінійок програмних продуктів (Software Product Lines, SPL) розглядається як стратегічний підхід до розробки, спрямований на радикальне зниження витрат, підвищення якісних показників та мінімізацію проектних ризиків. В основі SPL лежить принцип систематичного рекурентного використання спільних активів (Core Assets) для генерації сімейства програмних продуктів, що мають спільні архітектурні атрибути та функціональну поведінку. За результатами емпіричних досліджень, впровадження SPL дозволяє скоротити витрати на розробку гомогенних продуктів на величину, що перевищує 90%.

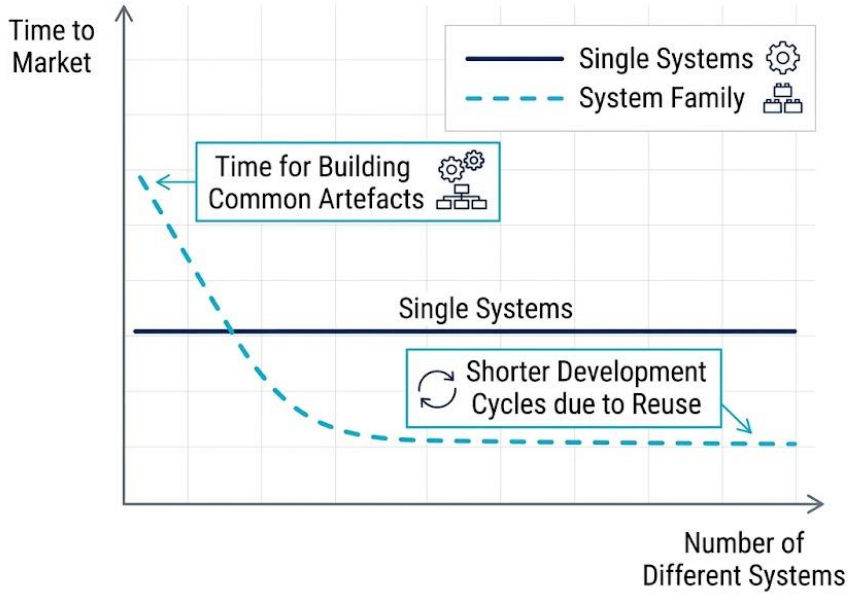


Рисунок 2.2 - Вплив парадигми SPL на динаміку Time-to-Market

Важливою перевагою підходу є динаміка часових витрат. На рисунку 2.2 продемонстровано, як після етапу інвестицій у створення та стабілізацію

основних активів час виходу наступних продуктів на ринок (Time-to-Market) суттєво скорочується порівняно з традиційними методами кастомної розробки.

Процес формування лінійки програмних продуктів базується на дворівневій моделі, що включає доменний інжиніринг (розробка «для повторного використання») та інжиніринг додатків (розробка «з використанням повторних активів»). Ключові етапи процесу включають:

- Доменний аналіз: ідентифікація спільних рис (commonalities) та варіативних особливостей (variabilities) у межах цільового сегмента ринку.

- Проєктування референтної архітектури: розробка гнучкого архітектурного каркаса, здатного підтримувати різні конфігурації.

- Керування варіативністю: забезпечення механізмів налаштування через параметризацію, успадкування або механізми динамічної лінковки (точки варіативності).

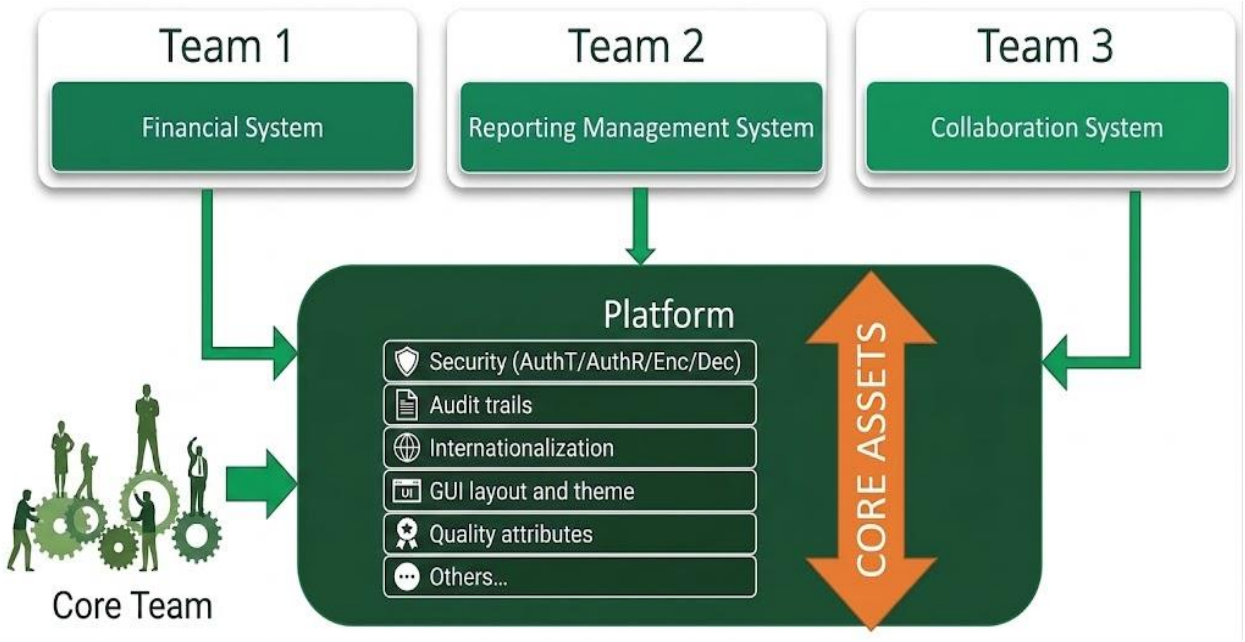


Рисунок 2.3 - Архітектурна модель SPL на прикладі інтегрованих бізнес-систем

На рисунку 2.3 наведено приклад архітектурної організації SPL, що об’єднує фінансові системи, модулі керування звітністю та сервіси

колаборації. У цій моделі спільна функціональність делегована в окремий рівень — Основні активи. Організаційна структура передбачає поділ на Команду основних активів, яка відповідає за розвиток платформи, та окремі продуктові команди, що фокусуються на специфічних потребах кінцевих користувачів.

Методологія SPL корелює з фундаментальними цілями програмної інженерії, зокрема щодо зменшення семантичного розриву між специфікацією вимог та програмним кодом, а також забезпечення прогнозованих атрибутів якості.

Для стандартизації підходу запропоновано фреймворк SPL (SPLFw). Згідно з цим фреймворком, лінійка програмних продуктів визначається як набір програмно-інтенсивних систем, що мають спільний керований набір функцій, задовольняють специфічні потреби сегмента ринку та розробляються зі спільного набору активів у регламентований спосіб.

Фреймворк SPLFw акумулює кращі галузеві практики та пропонує рекомендації щодо:

- Технічного та організаційного керування.

- Аналізу економічної ефективності та витрат.

- Методик доменного аналізу та архітектурного проектування.

Впровадження SPL забезпечує організаціям наступні конкурентні переваги:

- Масштабована продуктивність: можливість генерації великої кількості продуктів з мінімальними додатковими витратами.

- Зниження ризиків: використання перевірених, протестованих компонентів зменшує імовірність критичних дефектів.

- Гнучкість налаштувань: здатність оперативно адаптувати функціонал під вимоги конкретних місій за умови збереження цілісності системи.

Незважаючи на високу ефективність, перехід до SPL пов'язаний з певними викликами, які залишаються предметом актуальних дискусій:

- Високий поріг інвестицій: розробка основних активів потребує значних попередніх витрат часу та фінансів ще до виходу першого продукту.

- Дефіцит кваліфікації: проектування лінійок продуктів вимагає залучення архітекторів вищої кваліфікації з досвідом доменного моделювання.

- Стратегічне планування: успіх SPL залежить від готовності керівництва до довгострокових інвестицій у платформу, результати яких можуть бути неочевидними на коротких проміжках часу.

Таким чином, SPL є не просто технічним методом, а комплексною організаційною стратегією, яка дозволяє трансформувати процес розробки ПЗ у високотехнологічне виробництво з прогнозованою якістю та мінімальним часом виходу на ринок.

2.3. Порівняльний аналіз архітектурних парадигм програмних систем

Архітектурне проєктування становить фундамент процесу розробки програмного забезпечення, виступаючи сполучною ланкою між специфікацією функціональних вимог та програмною реалізацією системи. У межах сучасних інженерних процесів (зокрема каскадної моделі, Agile-методологій або компонентно-орієнтованої розробки) архітектура визначається як абстрактна, технологічно інваріантна модель елементів системи, їхніх взаємозв’язків та протоколів взаємодії. Ключова роль архітектурного підходу полягає у забезпеченні атрибутів якості — нефункціональних вимог, таких як надійність, масштабованість та безпека, а також у створенні базису для документування та оцінки системи.

У даному підрозділі розглядається еволюційна траєкторія архітектурних стилів, що охоплює перехід від монолітних структур до сервіс-орієнтованих, мікросервісних та нативних хмарних додатків.

Історично первинним підходом у комп'ютерній інженерії була монолітна архітектура, де всі компоненти системи інтегруються у єдиний виконуваний модуль. Такий підхід базується на концепції «редагування–компіляція–зв'язування» (edit-compile-link), що передбачає використання єдиного технологічного стеку та мови програмування для всього проєкту.

Незважаючи на відносну простоту розробки та розгортання на початкових етапах, монолітні системи мають суттєві обмеження:

- Технологічна ригідність: необхідність повної перекомпіляції та повторного розгортання всієї інстанції при внесенні мінімальних змін у код.

- Відсутність поліморфізму технологій: обмеження команди розробників єдиною мовою програмування.

- Складність масштабування: неможливість ізольованого горизонтального масштабування окремих високонавантажених компонентів.

- Ризики відмовостійкості: наявність єдиної точки відмови, де критична помилка в одному модулі призводить до деградації всієї системи.

Як механізм подолання дефіцитів монолітного підходу виникла сервіс-орієнтована архітектура (SOA). Вона передбачає декомпозицію системи на автономні сервіси, що взаємодіють у динамічному режимі за допомогою стандартизованих протоколів. Традиційно в межах SOA використовувалися XML-базовані технології, зокрема протоколи SOAP, WSDL та UDDI.

Проте широке впровадження SOA у сегменті малого та середнього бізнесу було обмежене через надмірну складність комунікаційних протоколів та значні накладні витрати обчислювальних ресурсів. Більше того, більшість сервісів у межах SOA продовжували використовувати спільні реляційні бази даних, що нівелювало переваги ізоляції та негативно впливало на загальну продуктивність і надійність.

Сучасна трансформація галузі, зумовлена розвитком IoT, смартфонів та стартап-екосистем, сформувала нові вимоги: швидкість виходу на ринок (Time-to-Market), мінімізація інфраструктурних витрат та економія масштабу в хмарних середовищах. Слід зазначити, що SOA є архітектурною моделлю

побудови ПЗ, тоді як SaaS (Software as a Service) — моделлю його надання кінцевому споживачеві.

Мікросервісна архітектура (MsA) постає як еволюційна відповідь на виклики сучасних хмарних обчислень. Це парадигма побудови додатків у вигляді набору малих, легкоатлантних компонентів, що взаємодіють через легкі протоколи обміну повідомленнями. Основним архітектурним стилем у межах MsA є REST (Representational State Transfer), що функціонує поверх протоколу HTTP з використанням формату JSON для серіалізації даних. Ключові характеристики мікросервісів:

- Повна автономність даних: кожен сервіс володіє власним сховищем даних, доступ до якого іншими модулями здійснюється виключно через програмні інтерфейси (API), що мінімізує рівень зв'язності (coupling).

- Незалежність розгортання та поліглотність: можливість використання оптимального технологічного стеку для кожного конкретного завдання.

- Ефективне масштабування: реплікація лише тих сервісів, що потребують додаткових ресурсів, за допомогою технологій контейнеризації (наприклад, Docker).

- Інноваційність та замінність: спрощення процесів тестування та можливість безболісної заміни окремих компонентів без порушення цілісності системи.

На відміну від SOA, мікросервісний підхід акцентує увагу на легкості комунікації та децентралізації даних. Проте перехід до MsA супроводжується зростанням складності процесів інтеграції, моніторингу та тестування розподілених систем, що потребує специфічних компетенцій від персоналу.

Для мінімізації ризиків та забезпечення стабільності життєвого циклу MsA критично важливою є інтенсивна автоматизація на основі наступних практик:

- CI/CD — безперервна інтеграція та доставка.

- TDD (Test-Driven Development) — розробка через тестування.

- Стандартизація структур проєктів для уніфікації інженерних підходів.

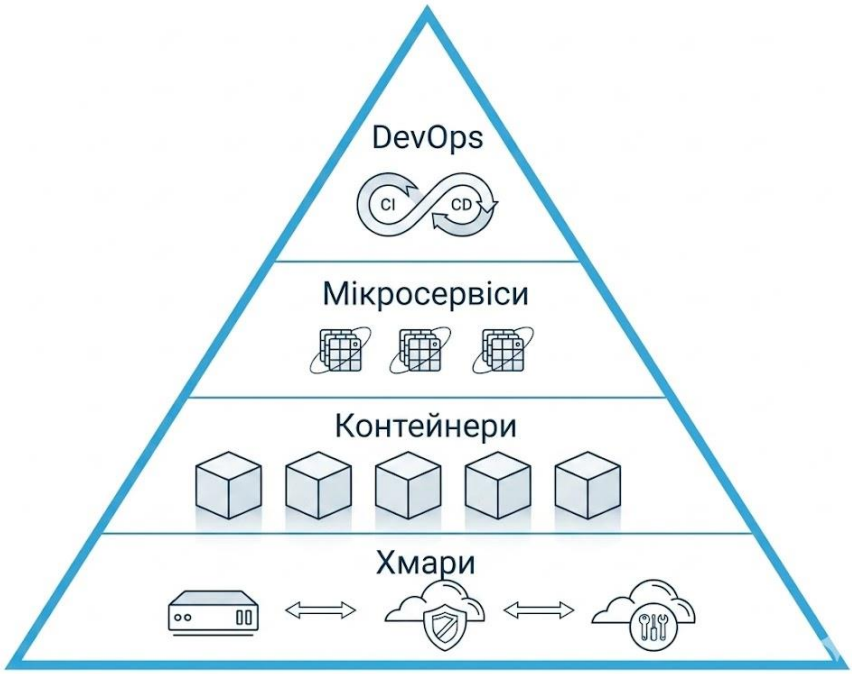


Рисунок 2.4 - Піраміда сучасних хмарних нативних застосунків

Концепція хмарно-орієнтованих додатків (CNA) уособлює вершину розвитку MsA, де портативність та повне використання переваг хмарної інфраструктури стають пріоритетними. CNA не залежать від конкретного хмарного провайдера та забезпечують автоматизовану оркестрацію, реєстрацію та моніторинг сервісів.

Сучасне екосередовище CNA базується на синергії мікросервісів, методології DevOps, контейнеризації та процесів безперервної доставки. Це створює адаптивне середовище, здатне динамічно реагувати на зміну навантаження та вимог ринку.

2.4. Детермінація ключових характеристик та атрибутів якості хмарно-орієнтованих застосунків

Хмарно-орієнтовані застосунки (Cloud-Native Applications, CNA) володіють специфічним набором характеристик, що зумовлюють їхню

архітектурну відмінність від традиційних локальних систем. Ці особливості визначають здатність системи до функціонування у динамічних, розподілених середовищах із високим ступенем невизначеності навантаження.

У традиційних архітектурах масштабування переважно розглядалося як вертикальне — збільшення обчислювальної потужності, обсягу оперативної пам'яті або дискового простору окремого вузла. Проте хмарна парадигма зміщує акцент на горизонтальне масштабування, що передбачає динамічне додавання нових вузлів до кластера.

1. Еластичність. На відміну від статичної масштабованості, хмарні системи мають підтримувати «масштабування вниз», що дозволяє вивільняти надлишкові ресурси в періоди низької активності, мінімізуючи операційні витрати.

2. Контейнеризація. Впровадження інструментів контейнеризації дозволяє уніфікувати процес розгортання та зробити горизонтальне масштабування економічно доступним не лише для великих корпорацій, а й для сегмента малого та середнього бізнесу. Це стало технологічним підґрунтям для мікросервісної архітектури.

3. Виклики проектування. Критичним аспектом залишається балансування між продуктивністю та вартістю. Недостатнє виділення ресурсів призводить до зростання латентності (latency), тоді як надмірне — до фінансової неефективності.

Сучасні хмарні рішення повинні забезпечувати підтримку широкого спектра фронтенд-технологій без модифікації логіки бекенд-рівня. Це зумовлено активним розвитком Інтернету речей (IoT) та мобільних платформ.

Архітектура має передбачати відокремлення представлення від даних (наприклад, через API Gateway), що дозволяє інтегрувати смартфони, інтелектуальні транспортні системи та промислові сенсори, які використовують специфічні протоколи (наприклад, MQTT).

Рисунок 2.5 – Хмарна модель білінгу, аудиту та моніторингу

Оскільки хмарні моделі базуються на принципах підписки та pay-as-you-go, система повинна мати вбудовані механізми білінгу та детального аудиту використання ресурсів.

Ефективний моніторинг є обов’язковою умовою дотримання параметрів якості (QoS) у режимі реального часу:

- Heartbeats та Health Checks: Безперервна перевірка життєздатності мікросервісів для оперативного реагування на збої.

- Візуалізація ресурсів: Необхідність у динамічних дашбордах, що відображають стан інфраструктури (Docker-контейнери, Kubernetes-поди) під час пікових навантажень.

Незважаючи на глобальну цифровізацію, забезпечення стабільного зв'язку в екстремальних умовах (віддалені наукові станції, пустелі, підземні споруди) залишається складним завданням.

Offline-first архітектура - функціонал хмарного додатка має дозволяти локальне виконання операцій з подальшою синхронізацією даних при відновленні з’єднання. Ця характеристика є критичною для систем управління проєктами, інструментів спільного редагування та автономних IoT-сенсорів, що працюють у важкодоступних регіонах.

У мультитенантних системах (наприклад, освітніх платформах або системах тестування знань) постачальник послуг підтримує єдину інстанцію коду для багатьох клієнтів (тенантів), створюючи для кожного ілюзію ізольованого середовища. Для забезпечення гнучкості без дублювання коду використовується концепція лінійок програмних продуктів (SPL). Це дозволяє моделювати варіативність функцій (наприклад, вибір мови інтерфейсу, теми оформлення чи специфічних модулів) під час виконання. Конфігурованість безпосередньо впливає на залучення клієнтів та знижує витрати на технічну підтримку.

Прийняття рішень щодо місця зберігання та обробки даних вимагає балансу між вартістю передачі трафіку та продуктивністю:

- Локальність даних: Зберігання даних ближче до клієнта (Edge computing) покращує взаємодію в додатках візуалізації.

- Централізація: Ефективна для складних аналітичних операцій, пошуку та фільтрації великих масивів даних.

- Якість обслуговування (QoS): Хмарна архітектура має запобігати проблемі «шумного сусіда» (noisy neighbor), де надмірне споживання ресурсів одним тенантом негативно впливає на інших учасників спільної інфраструктури.

Динамічна природа хмарних обчислень вимагає від архітекторів глибокого розуміння віртуалізованих середовищ, де традиційні методи проектування замінюються на принципи еластичності, автономності та високої обсервабільності.

2.5. Висновки по розділу

У межах другого розділу сформовано методологічний базис проєктування програмних систем, який враховує як функціональні, так і нефункціональні вимоги до сучасних застосунків. Встановлено, що архітектурні рішення є визначальним чинником забезпечення якості

програмного продукту, впливаючи на його продуктивність, масштабованість та надійність. Розглянута методологія лінійок програмних продуктів підтвердила ефективність використання підходів повторного застосування компонентів для зниження витрат і підвищення швидкості розробки. Проведений порівняльний аналіз архітектурних парадигм дозволив визначити найбільш доцільні підходи до створення хмарно-орієнтованих систем. У результаті сформовано систему ключових атрибутів якості, що слугує основою для проєктування ефективних і стійких до змін програмних рішень.

РОЗДІЛ 3. ПРОЕКТУВАННЯ ТА ІНЖЕНЕРНА РЕАЛІЗАЦІЯ ФРЕЙМВОРКУ

3.1. Концептуальна модель та архітектурна організація фреймворку

Intelligent Cloud є спеціалізованим архітектурним фреймворком, призначеним для швидкої розробки рисунок 3.1. (Rapid Application Development, RAD) дата-центричних хмарних систем. На відміну від вузькоспеціалізованих бібліотек, дане рішення пропонує комплексну систему, що складається з набору прикладних програмних інтерфейсів (API), багаторазових компонентів та хмарних сервісів, які інтегруються між собою для автоматизації типових операцій обробки даних.

Архітектура Intelligent Cloud характеризується високим ступенем адаптивності, що дозволяє застосовувати її як у монолітних, так і в мікросервісних конфігураціях. Запропоновані сервіси спроєктовані як автономні одиниці розгортання:

- Кожен сервіс здатен функціонувати як незалежна мікросервісна інстанція з власним ізольованим сховищем даних.

- За необхідності зниження латентності або спрощення інфраструктури, сервіси можуть бути інтегровані безпосередньо у єдиний процес через внутрішні виклики API.

Така гнучкість обумовлена складністю детермінації оптимального архітектурного стилю на ранніх етапах життєвого циклу програмного забезпечення. Як зазначається в [5], критичною помилкою багатьох проєктів є спроба побудови мікросервісної архітектури «з нуля», що часто призводить до надмірної складності без відповідної віддачі. Intelligent Cloud підтримує стратегію «спочатку моноліт», забезпечуючи безболісну декомпозицію системи на окремі частини в міру її зростання.

Ефективність Intelligent Cloud у контексті RAD забезпечується синергією двох підходів: розробки, керованої моделями (Model-Driven

Development, MDD), та підходу, керованого метаданими (Metadata-Driven Approach).

Оскільки більшість дата-центричних систем мають повторювану природу (операції CRUD, фільтрація, візуалізація), Intelligent Cloud використовує метадані сутностей для автоматичної генерації повного стеку програмних артефактів. Згенеровані компоненти суворо дотримуються патерну Model-View-Controller (MVC) і включають:

- Інтерфейс користувача (View): адаптивні форми та табличні представлення.

- Логічну модель (Model): структури даних та правила валідації.

- Контролер (Controller): обробники запитів та бізнес-логіку.

- Схему бази даних: SQL-скрипти для автоматичної ініціалізації середовища збереження даних.

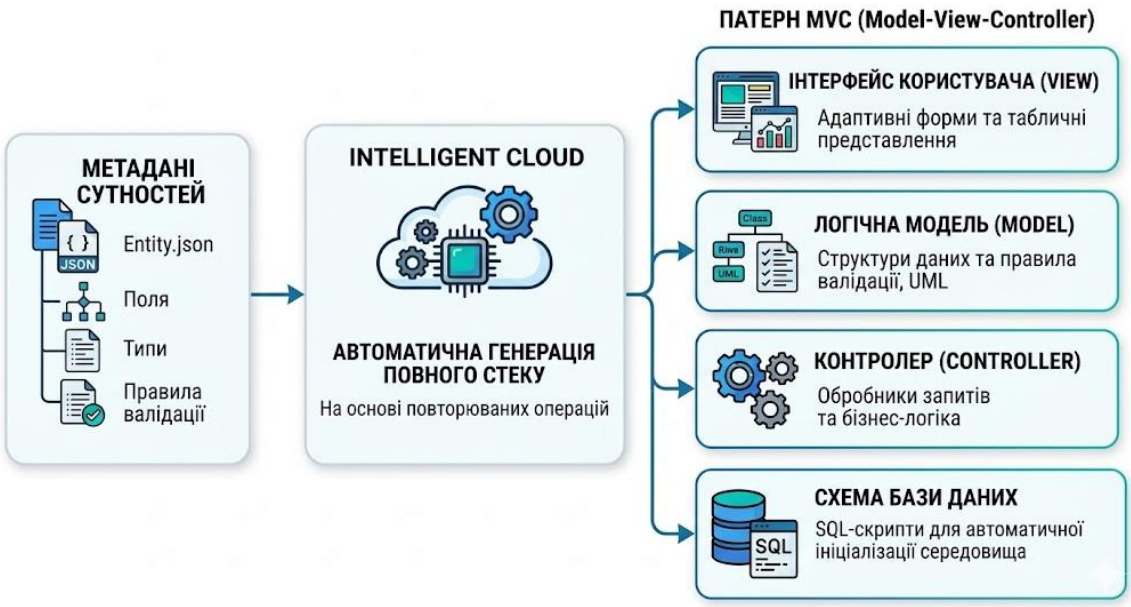


Рисунок 3.1 – Структура пропонованого фреймворку

Проте використання лише метаданих створює ризики щодо їх підтримки та масштабування. Для нівелювання цих ризиків у Intelligent Cloud інтегровано MDD-інструментарій, який дозволяє візуально моделювати архітектуру та

генерувати метадані автоматично, що значно підвищує консистентність системи.

Інтелектуальним ядром фреймворку є хмарна платформа Clowiz, яка виступає в ролі фасилітатора MDD-процесів. Вона дозволяє інженерам та аналітикам проєктувати складні хмарні системи з мінімальним залученням ручного кодування (Low-code/No-code парадигма).

Платформа Clowiz структурована за функціональним призначенням на три основні модулі:

- CodeGen: інструмент атомарної генерації, що створює окремі програмні одиниці (Java-класи, HTML-сторінки, скрипти міграції БД).

- FeatureGen: компонент високорівневої генерації, що формує цілісні функціональні модулі, забезпечуючи повну інтеграцію між рівнями MVC.

- AppGen: комплексний генератор додатків, що дозволяє трансформувати абстрактні моделі у повнофункціональні хмарні рішення за допомогою візуальних майстрів проєктування.

Застосування Intelligent Cloud у поєднанні з платформою Clowiz дозволяє не лише прискорити розробку, але й забезпечити високу якість архітектури, готову до горизонтального масштабування у хмарних середовищах.

3.2. Технологічні основи ORM у процесах автоматизованої генерації скриптів

У межах дата-центричних архітектур, до яких належить і описаний фреймворк Intelligent Cloud, фундаментальним завданням є подолання об'єктно-реляційного розриву. Механізми автоматичного мапування об'єктів (Object-Relational Mapping, ORM) виступають ключовим інструментом трансформації абстрактних логічних моделей ПЗ, описаних термінами об'єктно-орієнтованого програмування (класи, атрибути), у фізичні схеми

реляційних баз даних (таблиці, стовпці, зв'язки), що реалізуються через генерацію мови визначення даних (Data Definition Language, DDL).

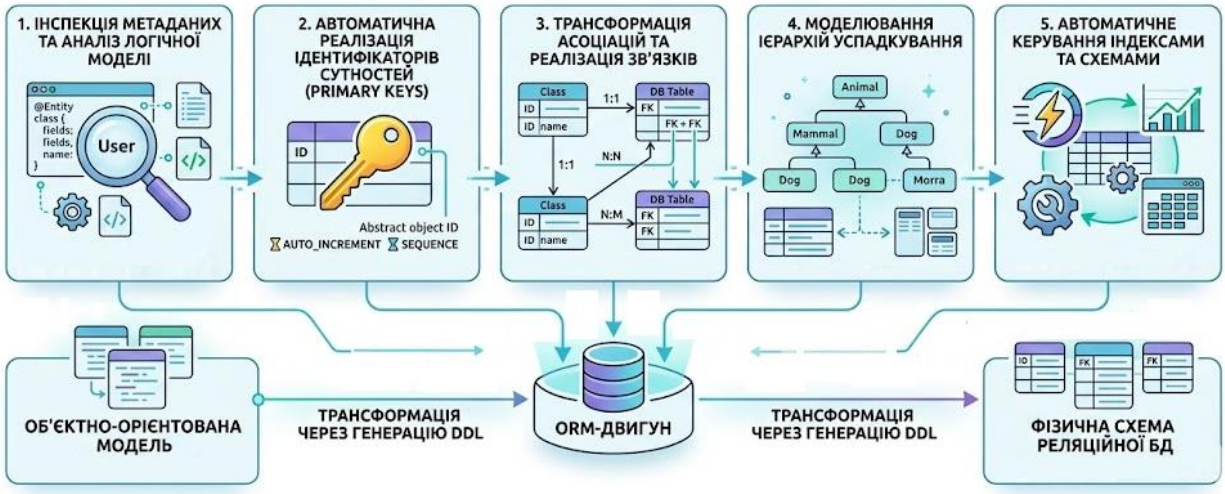


Рисунок 3.2 – Механізм автоматичного мапування об’єктів від логічної моделі до БД

Нижче наведено детальний аналіз ключових механізмів, які забезпечують точність та консистентність такої генерації:

1. Інспекція метаданих та аналіз логічної моделі

Процес генерації починається з аналізу метаданих логічної моделі (Model в MVC). ORM-двигун аналізує класи сутностей (Entities) та їхні анотації (або конфігураційні файли). На цьому етапі визначаються:

- Ідентифікація сутностей: класи, що мають бути відображені як таблиці БД (@Entity).

- Атрибути та типи даних: аналіз полів класу, їхніх типів (String, Integer, Date тощо) та відображення їх на відповідні типи діалекту SQL конкретної СУБД (наприклад, VARCHAR, INT, TIMESTAMP).

- Обмеження цілісності: аналіз анотацій, що визначають необов'язковість полів (nullable), унікальність (unique), довжину рядків (length).

Приклад: атрибут String name у класі з анотацією @Column(length=100, nullable=false) трансформується у визначення стовпця name VARCHAR(100) NOT NULL у DDL-скрипті.

2. Автоматична реалізація ідентифікаторів сутностей (Primary Keys)

ORM-механізм вимагає чіткого визначення первинного ключа для кожної таблиці. Під час генерації DDL аналізуються стратегії генерації ідентифікаторів:

- AUTO/IDENTITY: СУБД автоматично інкрементує значення (наприклад, SERIAL у PostgreSQL або AUTO_INCREMENT у MySQL).

- SEQUENCE: використання спеціальних об'єктів бази даних (Sequences) для генерації унікальних значень (поширено в Oracle, PostgreSQL).

- UUID: генерація універсальних унікальних ідентифікаторів на стороні додатка або БД.

3. Трансформація асоціацій та реалізація зв'язків

Це найбільш критичний етап ORM, де об'єктні посилання перетворюються на реляційні зв'язки. ORM-двигун під час генерації DDL реалізує наступні стратегії:

- Один-до-Одного (One-to-One) - реалізується через додавання зовнішнього ключа (Foreign Key, FK) в одну з таблиць із накладанням обмеження унікальності (UNIQUE) на цей FK стовпець.

- Багато-до-Одного (Many-to-One / One-to-Many) - реалізується через додавання FK-стовпця до таблиці на стороні «Багато». Це класична реалізація зв'язку між батьківською та дочірньою таблицями. Генерація DDL-скрипту включає не лише створення стовпця, а й визначення обмеження зовнішнього ключа, яке забезпечує посилальну цілісність.

Багато-до-Багатьох (Many-to-Many). Оскільки реляційна модель не підтримує прямий зв'язок Many-to-Many, ORM автоматично генерує проміжну таблицю зв'язків.

DDL-скрипт для цієї таблиці включає два стовпці, кожен з яких є зовнішнім ключем до відповідної основної таблиці. Комбінація цих двох стовпців зазвичай формує складений первинний ключ проміжної таблиці.

4. Моделювання ієрархій успадкування

Бази даних не мають вбудованого поняття успадкування класів. Для відображення ОО-успадкування в реляційну схему ORM-двигун під час генерації DDL застосовує одну з трьох стандартних стратегій:

- Одна таблиця на ієрархію:

Всі класи ієрархії (батьківський та дочірні) відображаються в одну велику таблицю.

До таблиці додається спеціальний стовпець-дискримінатор, який вказує, до якого саме типу належить поточний рядок.

DDL-скрипт генерує цю єдину таблицю з усіма можливими стовпцями для всіх класів ієрархії, де стовпці дочірніх класів зазвичай позначаються як nullable.

- З'єднані таблиці (Joined Table):

Кожен клас в ієрархії (включно з абстрактними) має власну таблицю.

Таблиці дочірніх класів містять лише специфічні для них атрибути, а також первинний ключ, який одночасно є зовнішнім ключем до таблиці батьківського класу.

DDL-скрипт генерує набір таблиць, пов'язаних Foreign Key обмеженнями на рівні первинних ключів.

- Таблиця на клас (Table per Concrete Class):

Для кожного конкретного (неабстрактного) класу генерується окрема таблиця.

Таблиця містить усі атрибути класу, включно з успадкованими.

DDL-скрипт генерує незалежні таблиці без прямих посилальних зв'язків між ними (для вибірки поліморфних запитів часто використовуються UNION операції).

5. Автоматичне керування індексами та схемами

Окрім генерації таблиць та зв'язків, сучасні ORM-двигуни автоматизують створення об'єктів для оптимізації продуктивності:

- Генерація індексів: автоматичне створення індексів для FK-стовпців (що є критичним для продуктивності операцій Join) та атрибутів, позначених як унікальні.

- Генерація схем (Schemas/Namespaces): підтримка організації таблиць у різних логічних схемах бази даних, що важливо для великомасштабних та мультитенантних систем.

Механізми автоматичного ORM у процесах генерації DDL-скриптів є критично важливими для Rapid Application Development (RAD). Вони гарантують сувору відповідність між логічною моделлю (Model) та фізичним сховищем даних, мінімізуючи ризик ручних помилок під час написання складних SQL-запитів визначення даних, спрощуючи еволюцію схеми та підвищуючи загальну надійність і підтримуваність дата-центричних хмарних систем.

3.3. Архітектурна організація фреймворку Intelligent Cloud

У даному підрозділі представлена високорівнева архітектура фреймворку Intelligent Cloud, візуалізована на рисунку 3.3, що відображає системну взаємодію основних структурних модулів. Архітектурне рішення базується на принципах модульності та ієрархічності, що дозволяє забезпечити гнучкість розробки дата-центричних систем.

Архітектура Intelligent Cloud декомпонована на чотири функціональні рівні (шари), кожен з яких виконує специфічні завдання в межах життєвого циклу розробки ПЗ:

- Рівень бібліотек та прикладних програмних інтерфейсів (Libraries & API):

- Даний шар акумулює набір багаторазових API, що інтегруються іншими рівнями системи для скорочення термінів розробки.

- Він включає інкапсульовані та верифіковані компоненти, зокрема утиліти загального призначення, модулі доступу до даних, веб-компоненти та інструментарій для керування метаданими.

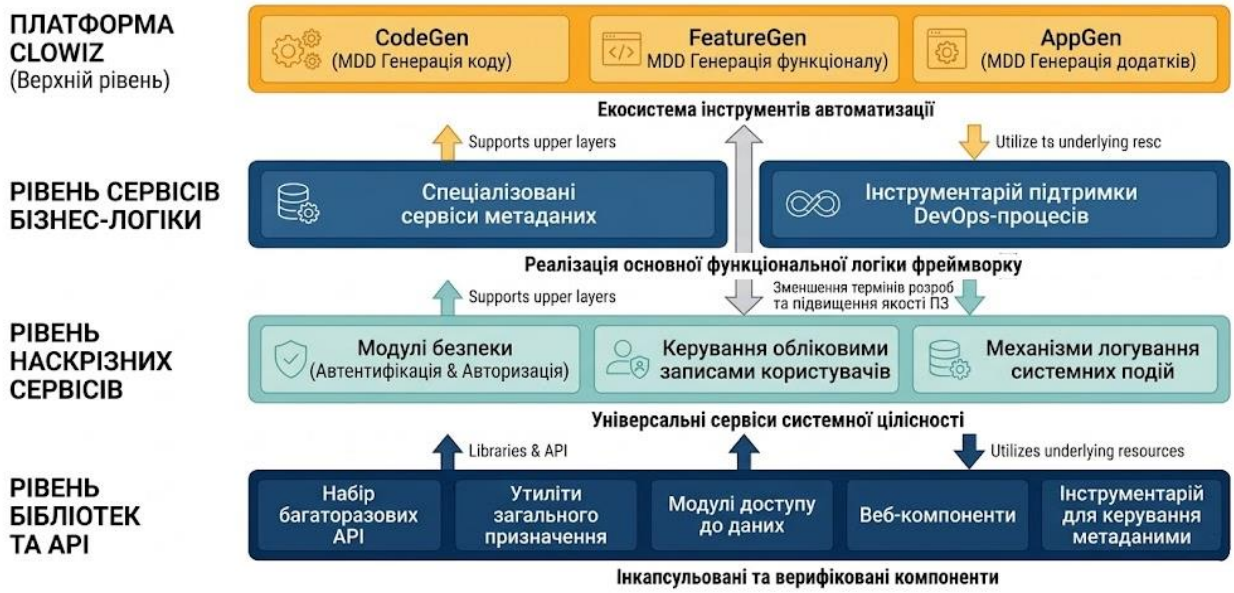


Рисунок 3.3 - Високорівнева архітектура фреймворку Intelligent Cloud

- Рівень наскрізних сервісів (Cross-cutting Services):

- Шар містить універсальні сервіси, що забезпечують системну цілісність та повторно використовуються сервісами додатків.

- До складу рівня входять модулі безпеки (автентифікація та авторизація), керування обліковими записами користувачів та механізми логування системних подій.

- Рівень сервісів бізнес-логіки (Application Logic Services):

- Рівень фокусується на реалізації основної функціональної логіки фреймворку.

- Включає спеціалізовані сервіси метаданих та інструментарій підтримки DevOps-процесів, що забезпечують безперервну інтеграцію та розгортання.

- Платформа Clowiz:

- Верхній рівень архітектури, представлений екосистемою інструментів автоматизації: CodeGen, FeatureGen та AppGen.

- Дані додатки реалізують парадигму розробки на основі моделей (MDD), автоматизуючи генерацію програмних артефактів.

Застосування фреймворку Intelligent Cloud в інженерії програмного забезпечення спрямоване на вирішення ряду критичних проблем розробки хмарних систем:

- оптимізація Time-to-Market: радикальне скорочення часових витрат на проєктування та імплементацію хмарно-орієнтованих рішень завдяки високому ступеню повторного використання компонентів.

- нівелювання ризиків дефіциту людських ресурсів: зниження залежності проєкту від наявності висококваліфікованих архітекторів та вузькоспеціалізованих розробників. Автоматизація рутинних операцій дозволяє залучати розробників середньої кваліфікації без втрати якості кінцевого продукту.

- концентрація на бізнес-цілях: надання розробникам можливості фокусуватися виключно на специфічній бізнес-логіці предметної області. Це досягається через делегування завдань побудови інфраструктури та базових функцій готовим сервісам, що надаються системою Intelligent Cloud «з коробки».

Стимулювання хмарної трансформації: заохочення організацій до прискореної міграції із застарілих локальних систем до нативних хмарних додатків. Такий перехід дозволяє повною мірою реалізувати переваги еластичності хмарних обчислень, підвищити операційну ефективність та оптимізувати сукупну вартість володіння IT-інфраструктурою.

З огляду на зазначене, Intelligent Cloud постає не лише як технічний інструментарій, а як стратегічна платформа для підвищення конкурентоспроможності розробки сучасних інформаційних систем.

3.4. Системний аналіз архітектурних, інженерних та експлуатаційних переваг фреймворку

Фреймворк Intelligent Cloud пропонує диференційований набір інструментів та сервісів, що мають високу прикладну цінність як для науковців, так і для практикуючих інженерів програмного забезпечення. Функціональні можливості системи класифікуються за п'ятьма ключовими доменами: архітектурна організація, технологія розробки, показники якості, користувацький досвід (UX) та операційна експлуатація.

З точки зору архітектурного проектування, Intelligent Cloud виступає фундаментальною основою для побудови різнорівневих дата-центричних хмарних екосистем. Гнучкість фреймворку дозволяє адаптувати його до різних категорій хмарних додатків, виходячи за межі суто інформаційних систем. Систематичне повторне використання прикладних програмних інтерфейсів (API), преконфігурованих компонентів та сервісів сприяє формуванню ефективної архітектури, мінімізуючи прояви антипатерну «reinventing the wheel» (винайдення велосипеда) та знижуючи рівень технічного боргу проекту.

У межах інженерного процесу впровадження Intelligent Cloud забезпечує наступні переваги:

- автоматизація генерації більшості типових програмних артефактів дозволяє розробникам сфокусувати зусилля виключно на реалізації специфічної логіки предметної області.

- наявність інтегрованих функцій, таких як підтримка мультитенантності та механізмів аудиту, суттєво підвищує продуктивність праці команди розробників.

- використання готових інструментаріїв знижує сукупні витрати на розробку та нівелює імовірність провалу проекту через відсутність вузькоспеціалізованих архітектурних компетенцій.

Високі якісні показники програмного забезпечення у межах Intelligent Cloud досягаються завдяки:

- всі базові компоненти фреймворку розроблені на основі розробки через тестування (Test-Driven Development), що гарантує високу відмовостійкість системи.

- генерація коду за допомогою платформи Clowiz забезпечує структурну консистентність проєкту. Будь-які оновлення або виправлення дефектів впроваджуються централізовано, що дозволяє миттєво масштабувати покращення на всі залежні компоненти системи.

З позиції системного адміністрування та експлуатації Intelligent Cloud надає розвинені можливості для моніторингу та підтримки:

- вбудовані механізми фіксації подій забезпечують повну прозорість функціонування системи.

- наявність преконфігурованих аналітичних дашбордів дозволяє експлуатаційним командам здійснювати оперативний моніторинг стану інфраструктури та сервісів, що є критично важливим для забезпечення стабільності хмарних рішень.

Таким чином, фреймворк Intelligent Cloud представляє собою цілісну екосистему, яка гармонізує вимоги щодо швидкості розробки, високої якості та операційної надійності сучасних хмарно-орієнтованих систем.

3.5. Висновки по розділу

У третьому розділі здійснено проєктування та реалізацію фреймворку Intelligent Cloud, що базується на принципах модульності, масштабованості та автоматизації процесів розробки. Розроблено концептуальну модель фреймворку, яка забезпечує узгоджену взаємодію його компонентів і підтримує розширюваність системи. Інтеграція ORM-технологій дозволила автоматизувати процеси роботи з базами даних, зменшивши складність розробки та підвищивши її ефективність. Проведений аналіз архітектурних та

інженерних рішень підтвердив доцільність використання обраного підходу при створенні хмарних застосунків.

Отримані результати свідчать про можливість суттєвого підвищення продуктивності розробки та якості програмних систем за рахунок використання запропонованого фреймворку.

РОЗДІЛ 4. ПРОЕКТУВАННЯ, АРХІТЕКТУРА ТА РЕАЛІЗАЦІЯ ФРЕЙМВОРКУ

У попередньому розділі було викладено теоретичні основи фреймворку Intelligent Cloud, представлено його високорівневу архітектуру, функціональні можливості, ключові характеристики, новизну, а також обґрунтовано вибір технологічного стеку та інфраструктурних рішень для всіх етапів імплементації.

Поточний розділ присвячено детальній розробці архітектури середнього та низького рівнів зазначеного фреймворку. Зокрема, наведено комплексний аналіз проектування програмних бібліотек, прикладних програмних інтерфейсів (API), сервісів та додатків, їхньої внутрішньої структури та механізмів взаємодії, що забезпечують цілісність фреймворку. Для формального моделювання архітектурних рішень широко застосовано мову уніфікованого моделювання UML. Крім того, висвітлено аспекти практичної реалізації та представлено репрезентативні зразки користувацького інтерфейсу фреймворку.

4.1. Деталізована архітектура фреймворку та опис компонентів

Архітектура фреймворку базується на багатошаровому підході, де кожен рівень групує компоненти за функціональною ознакою. Відповідно до концептуальних положень, ієрархія шарів має такий вигляд:

1. Бібліотеки та API.
2. Сервіси логіки додатків.
3. Наскрізні (крос-функціональні) сервіси.
4. Додатки платформи Clowiz.

На рисунку 4.1 наведено діаграму, що ілюструє архітектуру середнього рівня фреймворку Intelligent Cloud.

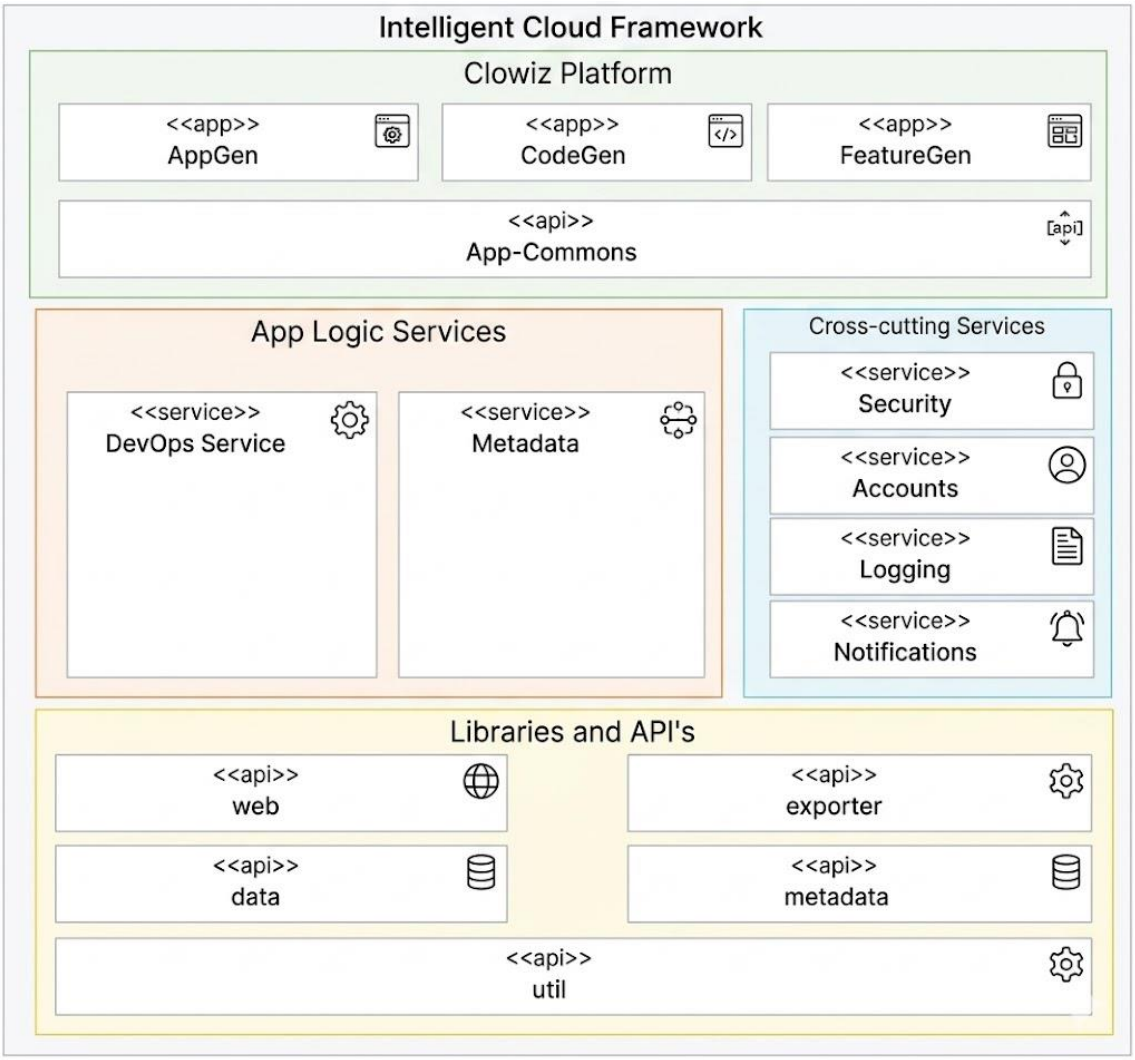


Рисунок 4.1 – Архітектура середнього рівня Intelligent Cloud

Рівень бібліотек та API акумулює набір бібліотек багаторазового використання, спрямованих на інтенсифікацію процесу розробки та підвищення продуктивності праці програмістів. Мінімізація ризиків виникнення програмних збоїв та гарантування надійності компонентів досягається завдяки тотальному впровадженню методології розробки через тестування (Test-Driven Development, TDD) для всього коду цього рівня.

Бібліотека універсальних утиліт містить сукупність універсальних компонентів, класів та методів, сумісних із будь-яким програмним забезпеченням на базі платформи Java. Структурно бібліотека охоплює понад 20 пакетів, що забезпечують реалізацію таких функціональних напрямків:

- стиснення даних;

- валідація даних;
- шаблонізація;
- кешування;
- криптографічний захист та безпека;
- інтеграція з поштовими сервісами;
- автоматизоване тестування.

Проектування більшості утиліт виконано за принципом забезпечення доступу до складної функціональності через атомарний виклик одного статичного методу. Наприклад, операції архівування та розархівування файлів реалізуються через прості виклики відповідних методів:

```
public static void unzip(InputStream in, String destDirectory)
public static Path unzip(InputStream in)
```

Компонент бібліотеки доступу до даних спроектовано для оптимізації та підвищення ефективності розробки модулів, що потребують персистентного зберігання та маніпулювання даними у реляційних базах даних (РБД). Бібліотека містить інструментарій, що підтримує дві основні парадигми взаємодії з БД:

1. Надійна та продуктивна підтримка виконання "чистого" SQL (plain SQL) безпосередньо всередині Java-додатків за допомогою стандарту JDBC.

2. Утиліти об’єктно-реляційного відображення (Object-Relational Mapping, ORM), що дозволяють здійснювати пряме декларативне зв'язування Java-класів із таблицями РБД без необхідності імплементації SQL-коду на рівні бізнес-логіки.

Імплементація цієї бібліотеки вирішує низку типових проблем, характерних для традиційних підходів до розробки шару доступу до даних:

- відсутність уніфікованого конфігураційного файлу для різних технік доступу;

- відсутність єдиного пулу ресурсів;

- складність керування життєвим циклом з'єднань, що призводить до витоків ресурсів через некоректне їх звільнення розробниками.

Прикладний програмний інтерфейс метаданих (Metadata API) є центральним компонентом фреймворку, що реалізує ядро функціональності, керованої метаданими (metadata-driven development). Він інкапсулює структури даних та логіку для роботи з метаописами.

Архітектурне рішення Metadata API представлено у вигляді діаграми класів UML на рисунку 4.2.

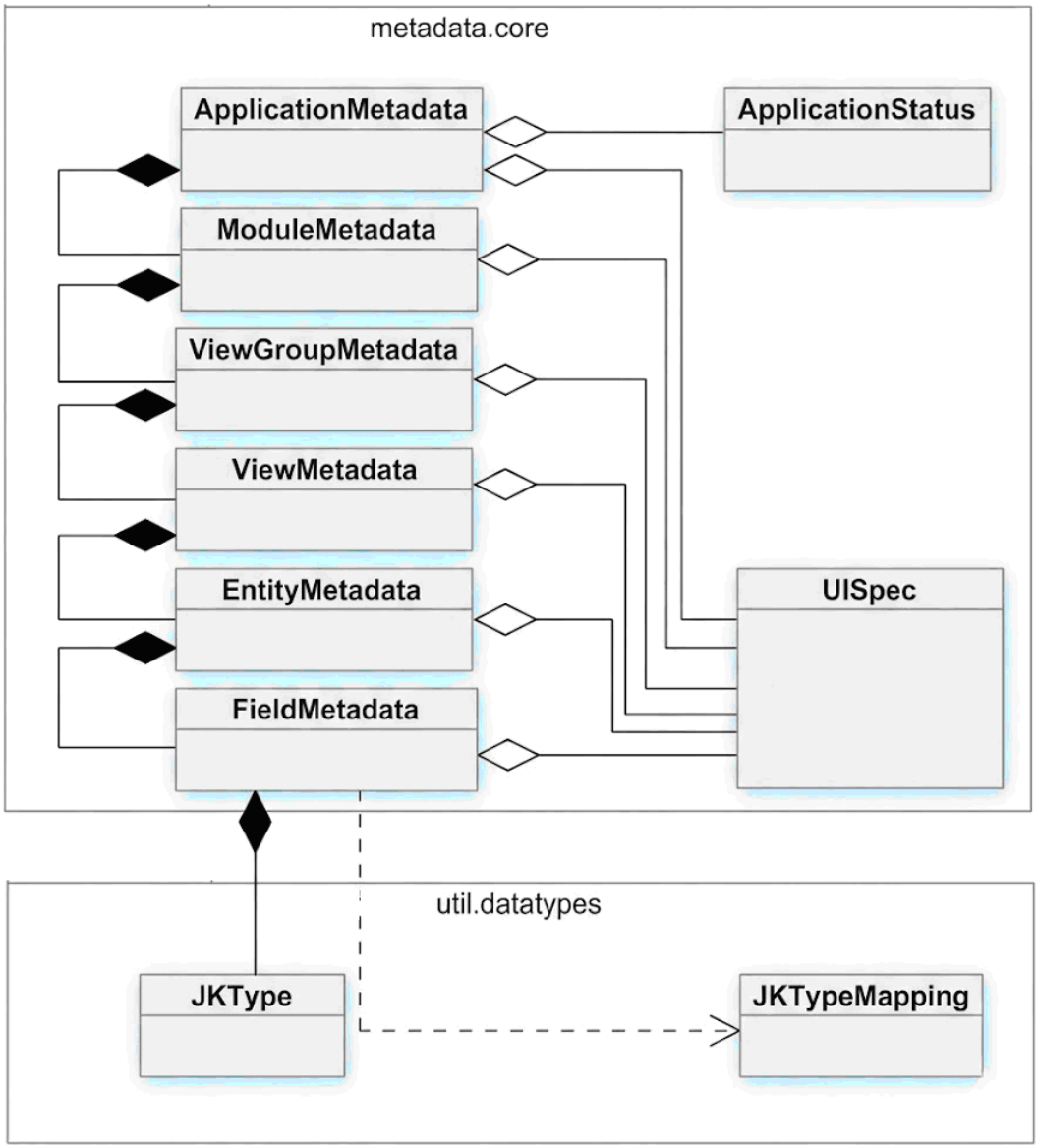


Рисунок 4.2 – Діаграма класів Metadata API

У таблиці 4.1 наведено систематизований опис функціонального призначення класів, представлених на діаграмі UML Metadata API (див. рис. 4.2).

Таблиця 4.1 - Опис класів Metadata API

Клас	Функціональний опис
ApplicationMetadata	Інкапсулює метадані рівня додатку в цілому (назва, статус, головна сторінка, локаль тощо). Виступає кореневим елементом ієрархії метаданих для модулів. Екземпляри цього класу застосовуються для повної генерації додатків та пов'язаних артефактів.
ModuleMetadata	Містить метадані логічного модуля (підсистеми). Слугує контейнером для метаданих груп перегляду. Використовується для генерації елементів навігаційного меню модулів у результуючих додатках, а також для класифікації та побудови ієрархічного дерева артефактів з метою забезпечення раціональної структури проекту.
ViewGroupMetadata	Описує метадані колекції представлень (наприклад, сукупності логічно пов'язаних сторінок). Екземпляри цього класу трансформуються в панелі меню при фінальній генерації додатку.
ViewMetadata	Містить метадані індивідуального представлення (екранної форми). Внутрішньо агрегує метадані сутності, необхідні для генерації артефактів патерну MVC (Model-View-Controller) для даного представлення.
EntityMetadata	Представляє універсальні метадані сутності, що можуть бути експортовані у різні формати вихідного коду (артефакти). Ці артефакти можуть включати представлення, контролери, моделі, скрипти створення таблиць БД тощо. EntityMetadata складається зі списку метаданих полів та визначення ідентифікатора.
FieldMetadata	Описує метадані окремих полів, що входять до складу метаданих сутності. Визначає ім’я поля, тип даних, обмеження цілісності (наприклад, можливість набувати значення null) тощо.
UISpec	Агрегує метадані, необхідні для генерації артефактів користувацького інтерфейсу. Включає параметри візуалізації, такі як ширина, висота, колір фону та колір тексту.
JKType	Інкапсулює інформацію щодо відображення (mapping) конкретного типу даних між різними технологічними доменами. Наприклад, може містити правила відповідності між типом даних varchar2 у SQL та класом String у Java.
JKTypeMapping	Реалізує реєстр відображень усіх типів даних, що використовуються у фреймворку, забезпечуючи конвертацію типів між усіма підтримуваними технологічними стеками.

API експортування (Exporter API) складається із сукупності класів, призначених для трансформації (експорту) екземплярів метаданих, створених

користувачами фреймворку за допомогою Metadata API, у фінальні програмні артефакти вихідного коду. На рисунку 4.3 представлена діаграма пакетів, що ілюструє структуру Exporter API.

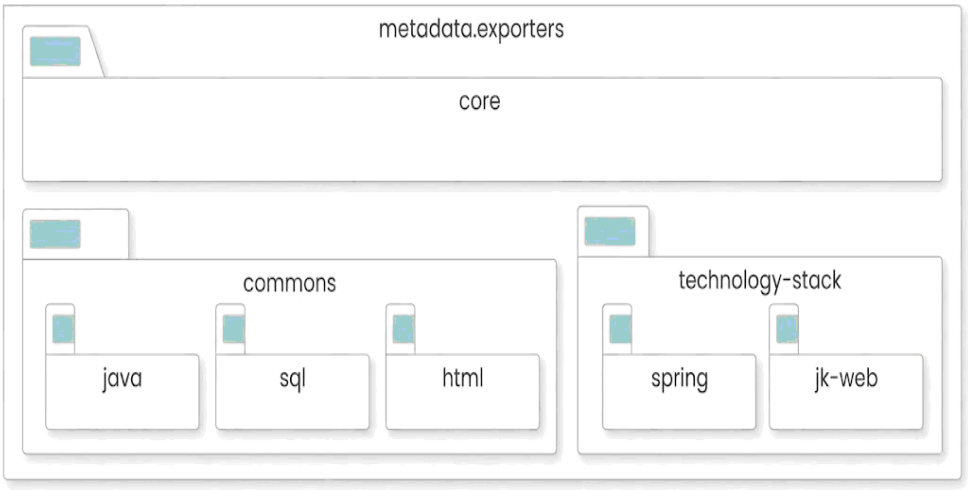


Рисунок 4.3 – Діаграма пакетів Exporter API

На рисунку 4.4 деталізовано вміст базового пакета (`core`) цього API. Пакет включає ключові інтерфейси та класи експортерів, такі як `ApplicationExporter` та `EntityMetadataExporter`. Реалізація цих інтерфейсів є необхідною умовою для забезпечення експорту метаданих у нові додатки для конкретного стеку технологій, а також для генерації їхніх внутрішніх артефактів.

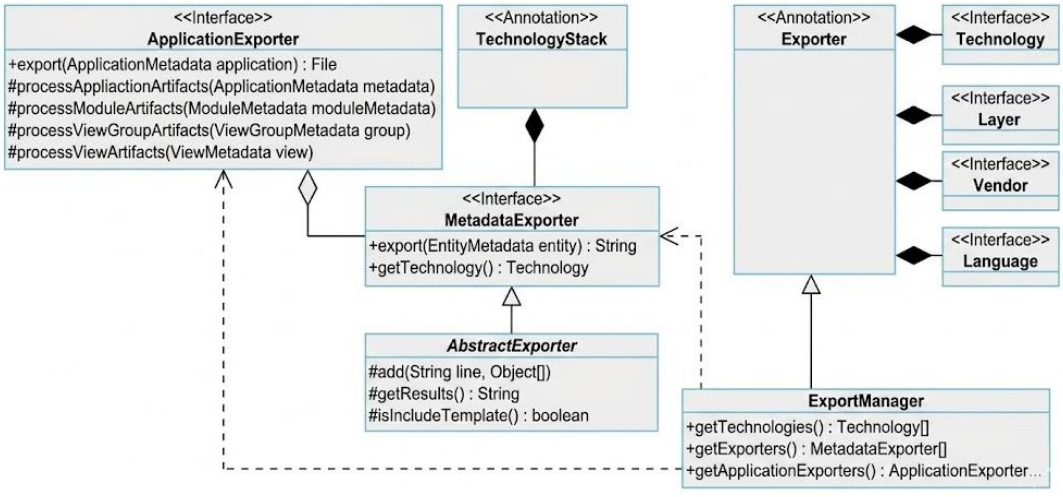


Рисунок 4.4 – Базовий пакет (core) Exporter API

Рисунок 4.5 демонструє вміст пакета загальних компонентів (`commons`) Exporter API. Цей пакет включає спільні класи експортерів, що використовуються в різних технологіях, та ілюструє їхній зв'язок із базовим пакетом (`core`).

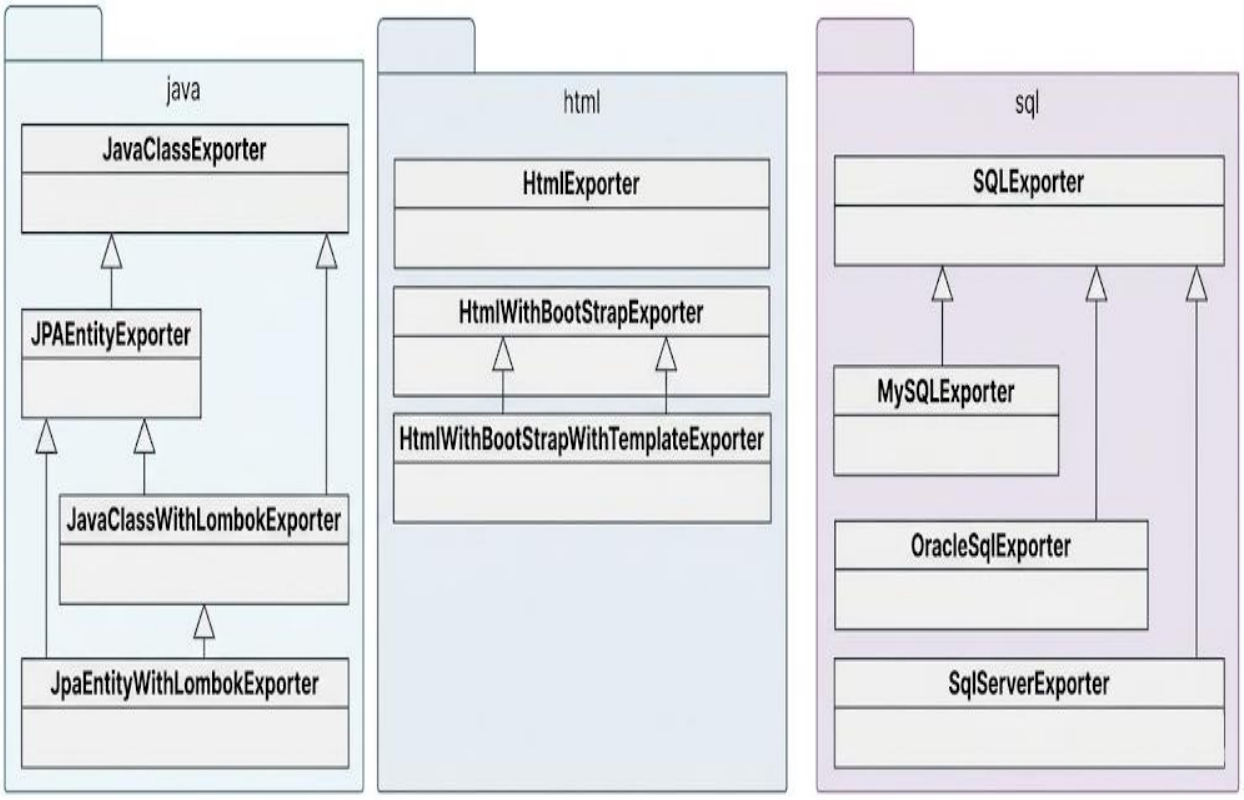


Рисунок 4.5 – Пакет загальних компонентів (commons) Exporter API

Кожен конкретний експортер у цьому пакеті призначений для генерації індивідуальних програмних артефактів, таких як:

- класи Java;
- сторінки HTML;
- SQL-скрипти мови визначення даних (DDL) для створення таблиць.

На рисунку 4.6 представлено пакет технологічних реалізацій (`technologies`). На поточному етапі розробки фреймворк підтримує генерацію додатків для стеків JavaEE та Spring Framework.

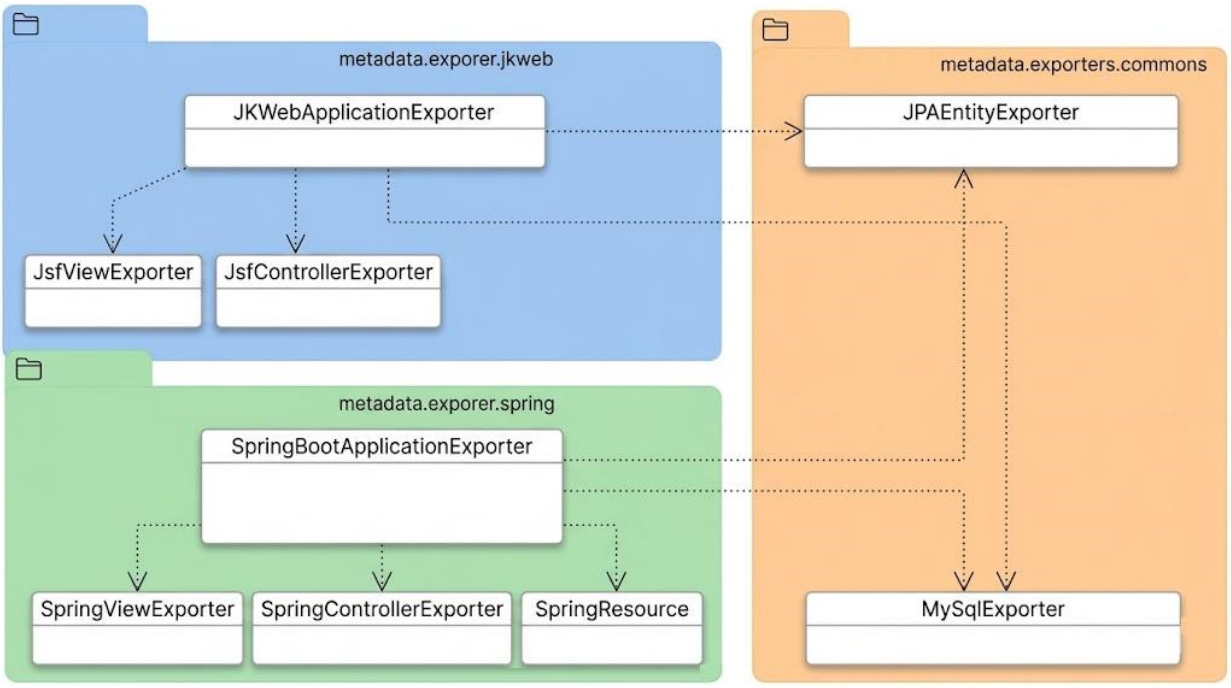


Рисунок 4.6 – Пакет технологічних реалізацій (technologies) Exporter API

4.2. Функціонально-логічна архітектура та життєвий цикл хмарних систем фреймворку Intelligent Cloud

Модуль AppGen позиціонується як верхній рівень екосистеми Intelligent Cloud, що реалізує концепцію розробки без написання коду (No-code development). Даний інструментарій забезпечує повний цикл інженерії програмного забезпечення: від концептуального проєктування та візуального моделювання до автоматизованого розгортання в хмарному середовищі.

4.2.1. Методологія проєктування на основі метаданих

В основі функціонування AppGen лежить парадигма розробки, керованої моделями (Model-Driven Engineering, MDE). Користувач детермінує архітектурні, логічні та інтерфейсні аспекти майбутнього додатка виключно через конфігурацію метаданих. Це дозволяє абстрагуватися від низькорівневої імплементації та зосередитися на бізнес-логіці предметної області.

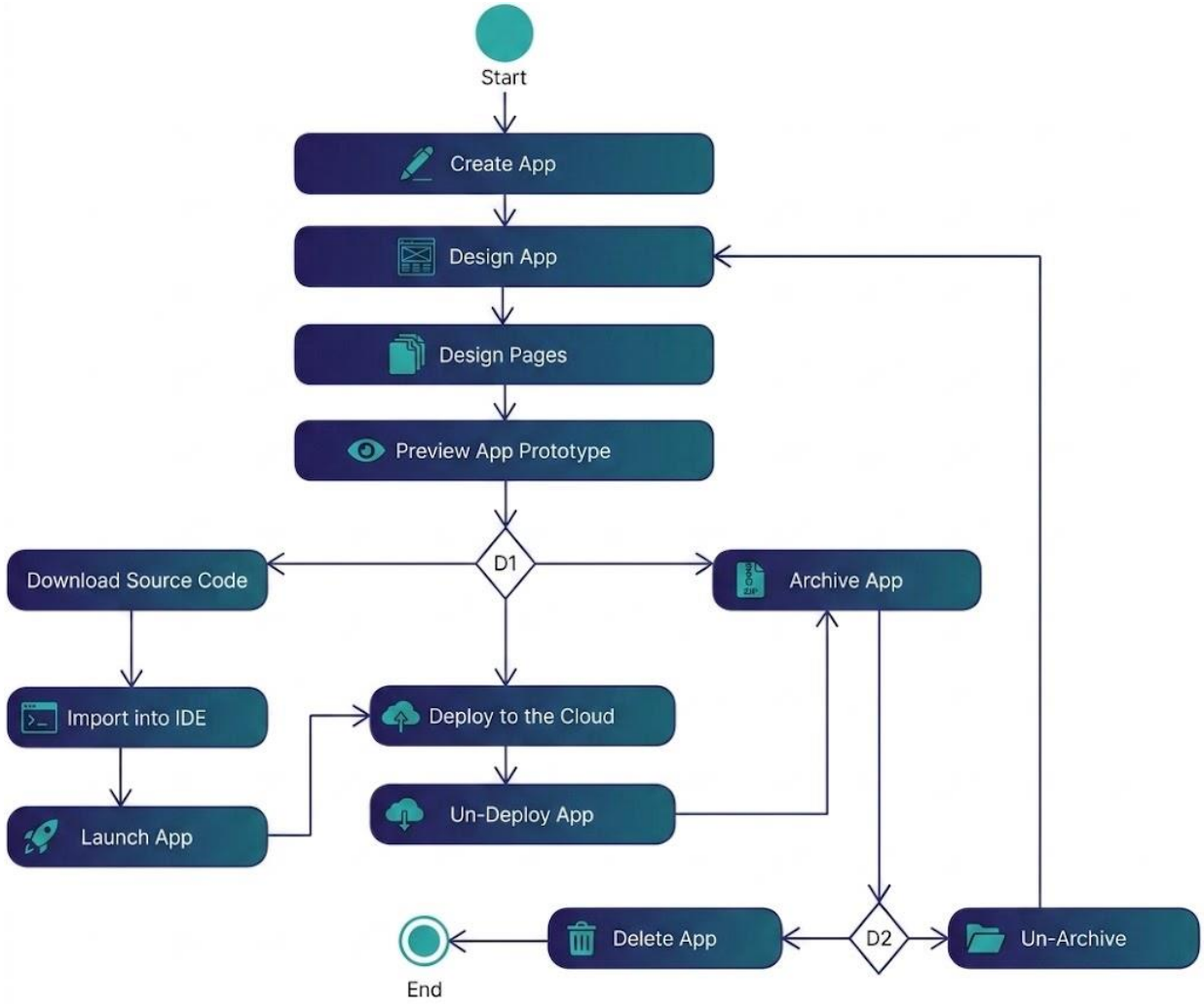


Рисунок 4.10 - Процес розробки в AppGen

Процесна модель розробки в середовищі AppGen деталізована на діаграмі діяльності (Activity Diagram), що наведена на рисунку 4.10. Вона демонструє ітеративний характер проєктування, включаючи етапи верифікації метаданих та попереднього перегляду інтерфейсів перед генерацією артефактів.

Ця схема діяльності (рис. 4.10) описує повний життєвий цикл розробки та управління хмарною No-code програмою в "AppGen", що є частиною платформи Intelligent Cloud. Користувач розпочинає створення програми, виконуючи крок Create App (Створити програму). Після цього починається ітеративний процес проєктування. Користувач переходить до Design App (Спроєктувати програму), щоб визначити її загальну архітектуру та логіку, а

потім до Design Pages (Спроєктувати сторінки), щоб деталізувати користувацький інтерфейс.

Коли проєктування завершено, користувач може виконати Preview App Prototype (Попередній перегляд прототипу програми), щоб оцінити, як програма виглядає та працює. На цьому етапі користувач опиняється перед важливим вибором (ромб прийняття рішень), де він може обрати один із трьох основних шляхів:

- перший шлях орієнтований на локальну розробку: користувач обирає Download Source Code (Завантажити вихідний код), потім Import into IDE (Імпортувати в IDE) і, нарешті, Launch App (Запустити програму) на локальному комп'ютері. Цей шлях є самодостатнім і завершує локальну активність.

- другий шлях веде до пряме заархівування програми через крок Archive App (Архівувати програму), якщо робота над нею призупинена.

- третій, основний хмарний шлях, — це виведення програми в продакшн. Користувач обирає Deploy to the Cloud (Розгорнути у хмарі), після чого програма стає активною у Intelligent Cloud.

Якщо в майбутньому програма в хмарі більше не потрібна, користувач може виконати крок Un-Deploy App (Скасувати розгортання програми). Після цього, або якщо програма була заархівована безпосередньо, вона перебуває в архівному стані (Archive App).

Останнім етапом управління є прийняття рішення щодо заархівованої програми. Користувач може обрати Un-Archive (Розархівувати), що повертає його назад у цикл проєктування, до кроку Design App. Альтернативним, кінцевим варіантом є Delete App (Видалити програму). Після виконання цього кроку життєвий цикл програми завершується (вузол End).

4.2.2. Модель станів та життєвий цикл додатка

Життєвий цикл інформаційних систем, створених засобами AppGen, регламентується формалізованою моделлю переходів станів. Кожна інстанція додатка проходить через наступні ключові фази:

- Фаза розробки (In Development Phase): Початковий стан, у якому здійснюється маніпулювання метаданими, проєктування бази даних та налаштування компонентів MVC.

- Розгорнутий стан (Deployed): Результат успішної трансляції метаданих у виконуваний код та ініціалізації середовища виконання.

- Архівний стан: Тимчасова деактивація додатка зі збереженням цілісності всіх налаштувань для можливості подальшої реінкарнації або остаточного видалення.

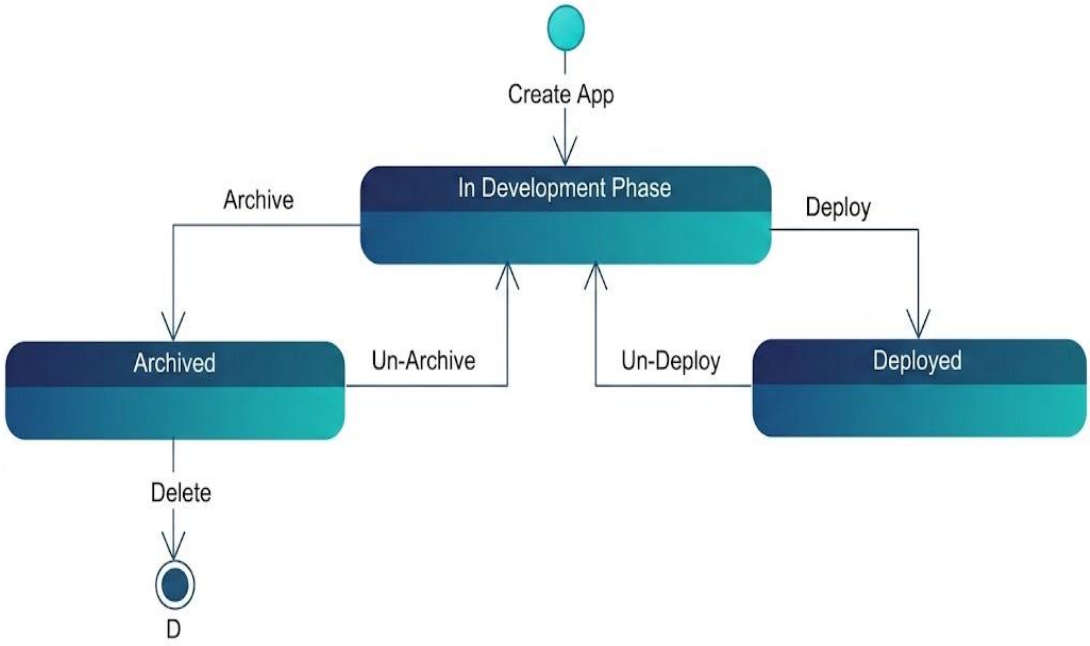


Рисунок 4.11 - Робочий процес статусів AppGen

Система підтримує реверсивні переходи, зокрема можливість скасування розгортання (undeploy) для повернення продукту на ітерацію доопрацювання. Візуалізація повного робочого процесу (workflow) та логіки трансформації станів представлена на рисунку 4.11.

4.2.3. Персистентність та інваріантність рівня збереження даних

Для гарантування цілісності та довговічності результатів проєктування, сервіси AppGen інтегровані з реляційною базою даних. Взаємодія здійснюється через абстрактний шар Data API, що входить до складу бібліотечного рівня фреймворку Intelligent Cloud.

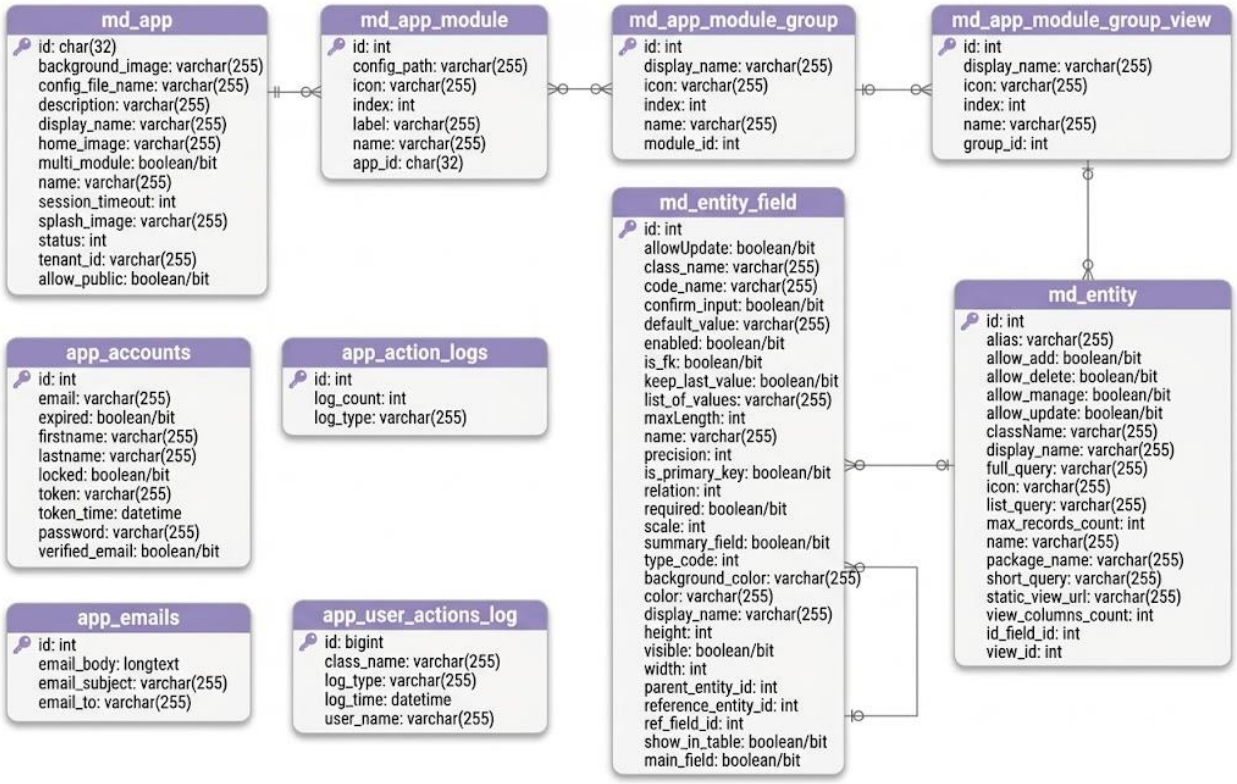


Рисунок 4.12 - Діаграма «сутність-зв’язок» (ER-діаграма)

Поточна технічна реалізація базується на СУБД MySQL, проте архітектура рішення забезпечує повну технологічну незалежність. Завдяки використанню специфікацій JDBC (Java Database Connectivity), система підтримує горизонтальну сумісність із будь-якими реляційними СУБД (PostgreSQL, Oracle, SQL Server) без необхідності рефакторингу вихідного коду. Структурна організація бази даних та ієрархія сутностей фреймворку відображені на повній діаграмі «сутність-зв’язок» (ERD), наведеній на рисунку 4.12.

Впровадження модуля AppGen у межах платформи Intelligent Cloud дозволяє радикально знизити когнітивне навантаження на розробників та

скоротити час виходу продукту на ринок (Time-to-Market), зберігаючи при цьому високу консистентність та надійність хмарних додатків.

4.3. Розробка структури фреймворку

Рисунок 4.13 демонструє комплексну архітектуру Intelligent Cloud — RAD-фреймворку, призначеного для швидкої розробки хмарних застосунків на основі моделей. Структура системи організована у вигляді ієрархічних рівнів, що забезпечують шлях від фундаментальних компонентів до готового розгорнутого рішення. Структура фреймворку побудована як п'ятишарова архітектура зверху вниз:

1. Розгортання — дві рівноправні конфігурації (моноліт або мікросервіси), між якими можна переходити без переписування коду — це ключова ідея "спочатку моноліт".

2. MVC-стек — чотири типи артефактів, що автоматично генеруються: View (HTML-форми, таблиці), Model (Java Entity, валідація), Controller (REST, бізнес-логіка) та DB Schema (SQL/Flyway).

3. Метадані сутностей (MDD/MDE) — центральний вузол, що живить генерацію вгору (MVC) і спрямовує роботу інструментів Clowiz вниз. Визначаються поля, типи, зв'язки та архітектурний шаблон.

4. Платформа Clowiz — три модулі з наростаючою абстракцією: CodeGen → окремі файли, FeatureGen → цілісні MVC-модулі, AppGen → повний застосунок через візуальні майстри.

5. Основа — набір API, багаторазових компонентів і хмарних сервісів як автономних одиниць розгортання.

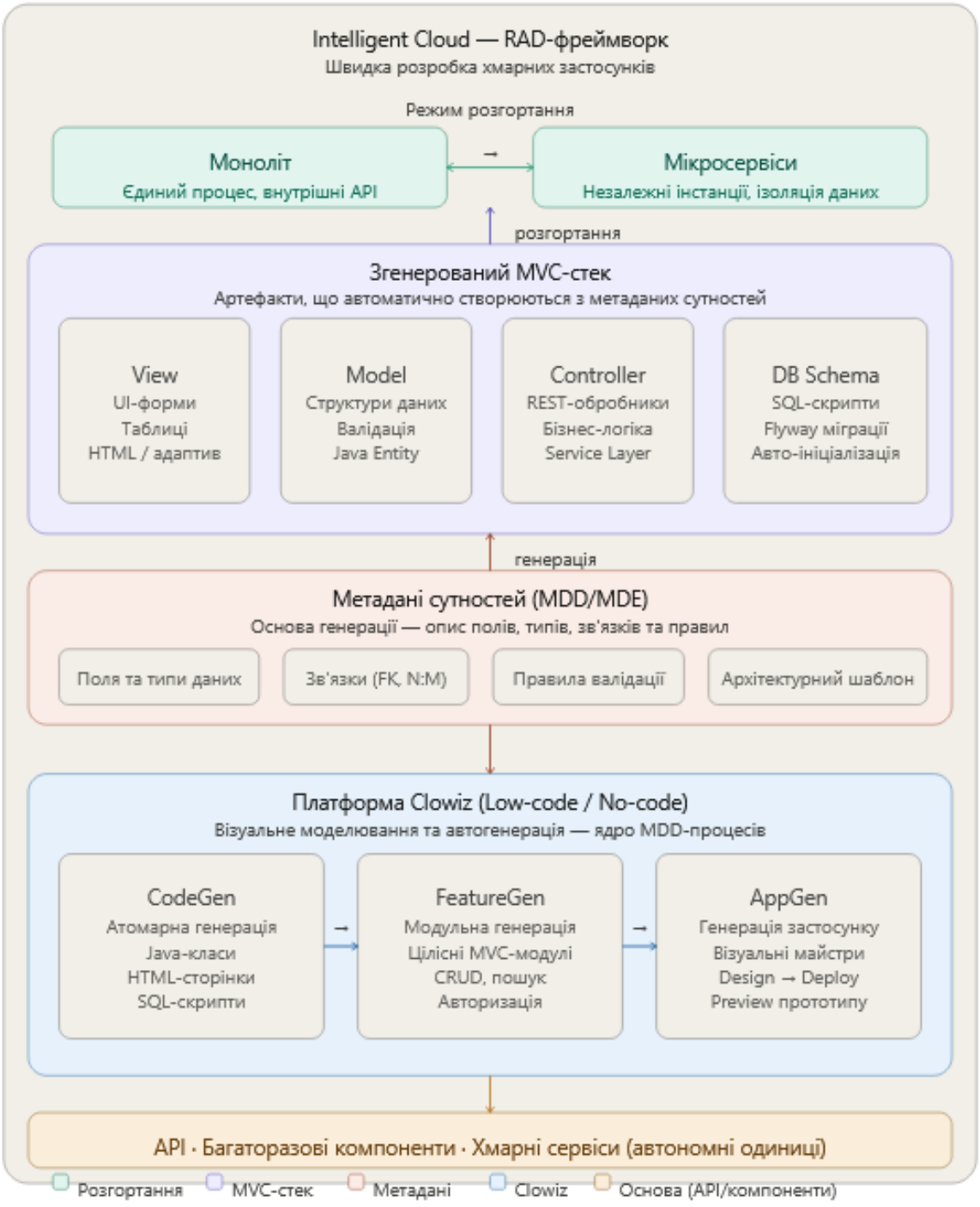


Рисунок 4.13 – Структура фреймворку

Процес розробки у фреймворку має лінійну спрямованість: від визначення метаданих через інструменти Clowiz до створення MVC-артефактів та фінального розгортання у вибраному режимі.

4.4. Представлення інтерфейсу пропонованого фреймворку

На рисунку 4.14 подано інтерактивний інтерфейс фреймворку Intelligent Cloud. Навігація по бічній панелі:

- CodeGen — визначення сутності (поля, типи, PK/FK), вибір що генерувати (Entity, Controller, View, SQL), попередній перегляд форм і таблиць, вивід Java-коду. Є три вкладки: Сутність → Перегляд → Код.

- AppGen — керування застосунками (хмара / локально / архів), візуалізація циклу розробки з 5 кроків.

- FeatureGen — генерація повних модулів (CRUD, пошук, авторизація) з прогрес-барами по кожному шару MVC.

- Сутності — реєстр усіх entity з зв'язками (1:N, N:M) та кнопками переходу до CodeGen.

- REST API — автозгенерований список ендпоінтів з методами (GET/POST/PUT/DELETE).

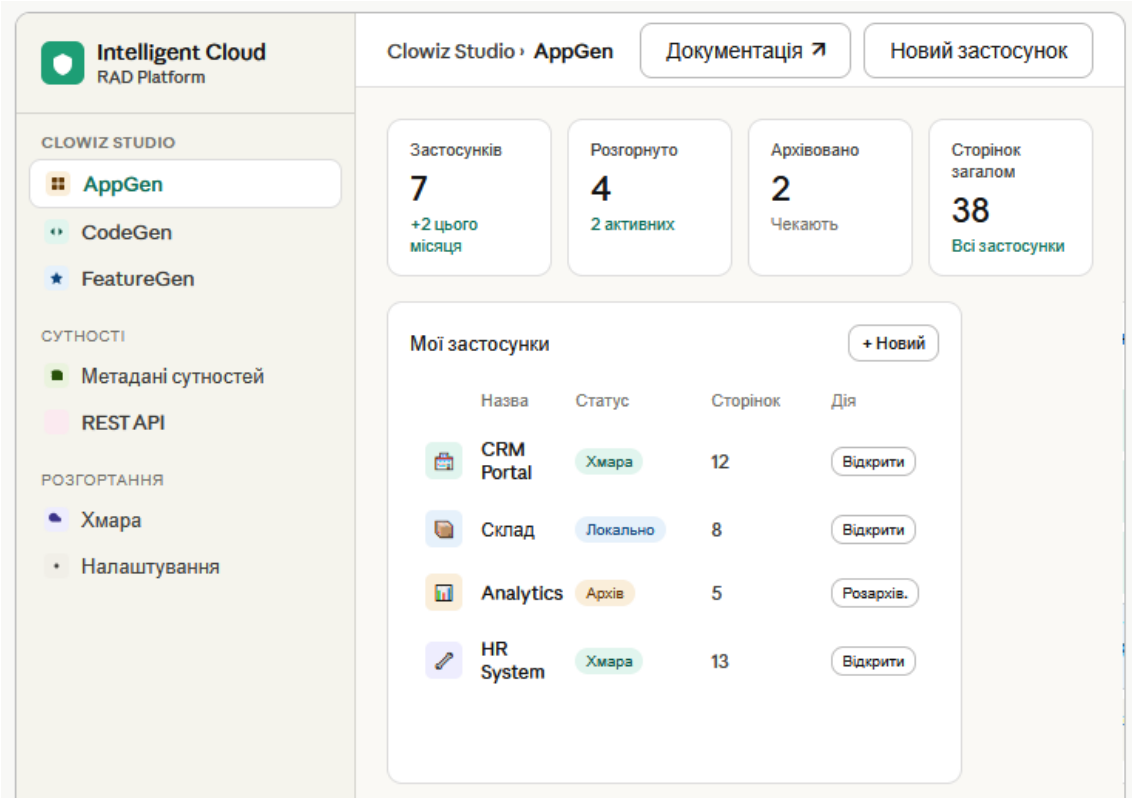


Рисунок 4.14 – Головна сторінка (вкладка AppGen)

Рисунок 4.14 демонструє головну панель керування (Dashboard) модуля AppGen, який є частиною платформи Intelligent Cloud. Це центральний вузол для моніторингу та управління всіма проєктами користувача, де візуалізовано ключові метрики розробки та стан розгорнутих систем.

Нижче наведено детальний опис структурних блоків інтерфейсу:

1. Навігаційна панель (Бічне меню)

Зліва розташовано деревоподібне меню, що відображає архітектурну логіку платформи:

- CLOWIZ STUDIO: Вибір між інструментами генерації (AppGen, CodeGen, FeatureGen). Наразі обрано AppGen, що підсвічується активним станом.

- СУТНОСТІ: Швидкий доступ до керування метаданими сутностей та налаштуваннями REST API.

- РОЗГОРТАННЯ: Розділи для моніторингу хмарної інфраструктури та конфігурації системних параметрів.

2. Панель статистичних показників (KPI)

Верхня частина робочої області містить чотири інформаційні картки, які дають миттєве уявлення про масштаб роботи:

- Застосунків: Загальна кількість створених проєктів із індикатором динаміки (+2 за поточний місяць).

- Розгорнуто: Кількість систем, що пройшли процес деплою, з уточненням про наявність 2-х активних інстанцій.

- Архівовано: Проєкти, що перебувають у режимі очікування або тимчасового зберігання.

- Сторінок загалом: Сумарна кількість згенерованих інтерфейсних форм у всіх застосунках.

3. Реєстр застосунків (Мої застосунки)

Центральна таблиця дозволяє керувати життєвим циклом конкретних продуктів:

- Індикація статусів: Кожен застосунок має кольорове маркування свого стану:

- Хмара: Система активна та розгорнута на сервері (наприклад, CRM Portal, HR System).

- Локально: Проєкт перебуває у фазі розробки або локального тестування (Склад).

- Архів: Проєкт виведено з експлуатації зі збереженням даних (Analytics).

- Контекстні дії: Кнопки праворуч дозволяють миттєво перейти до редагування («Відкрити») або відновити проєкт із архіву («Розархів.»).

4. Елементи швидкої дії

У правому верхньому куті розташовано функціональні кнопки для оперативного доступу до документації та ініціалізації процесу створення нового проєкту («Новий застосунок»).

На рисунках 4.15 – 4.17 подано інтерфейс CodeGen — одного з ключових модулів платформи Intelligent Cloud. Це сучасне Low-code середовище, де розробник може проектувати архітектуру застосунку на рівні метаданих, а система автоматично перетворює їх на готовий програмний код.

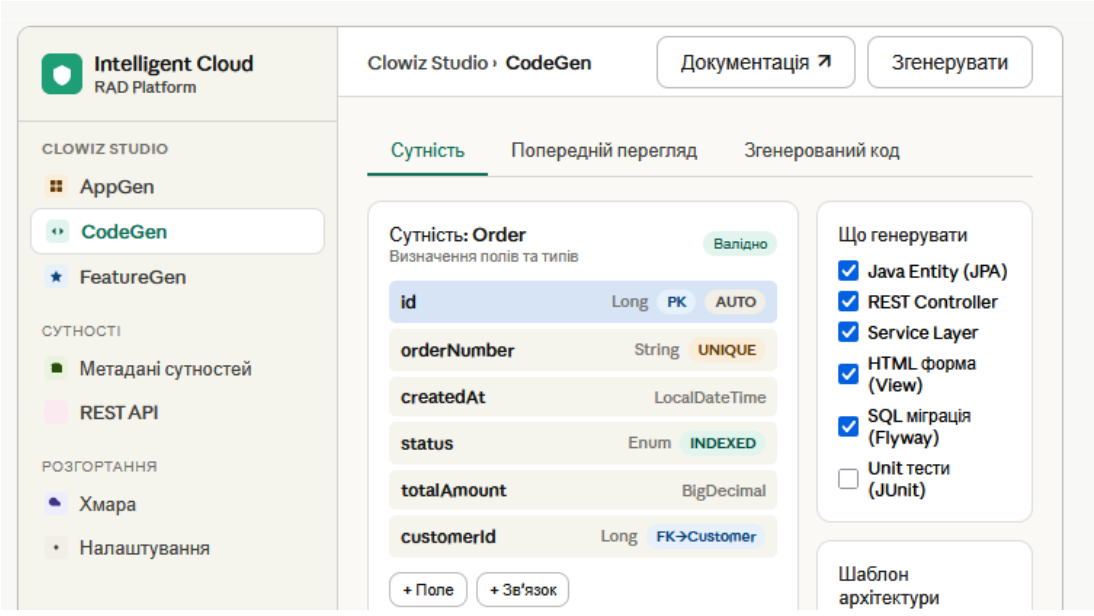


Рисунок 4.15 – Модуль CodeGen (сутність)

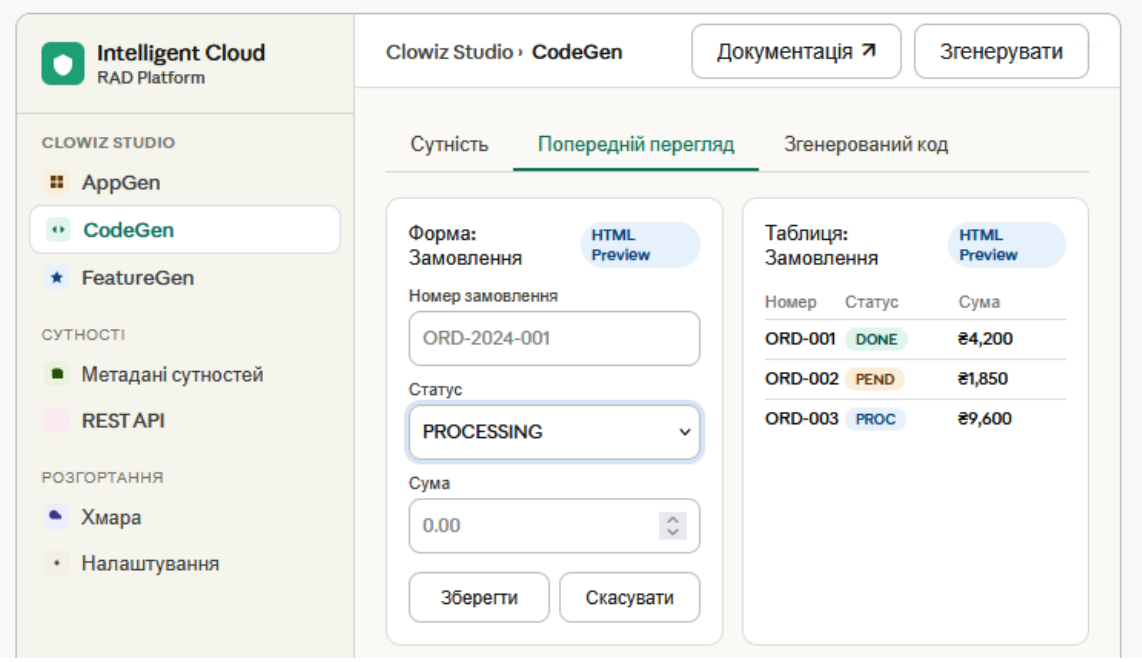


Рисунок 4.16 – Модуль CodeGen (попередній перегляд)

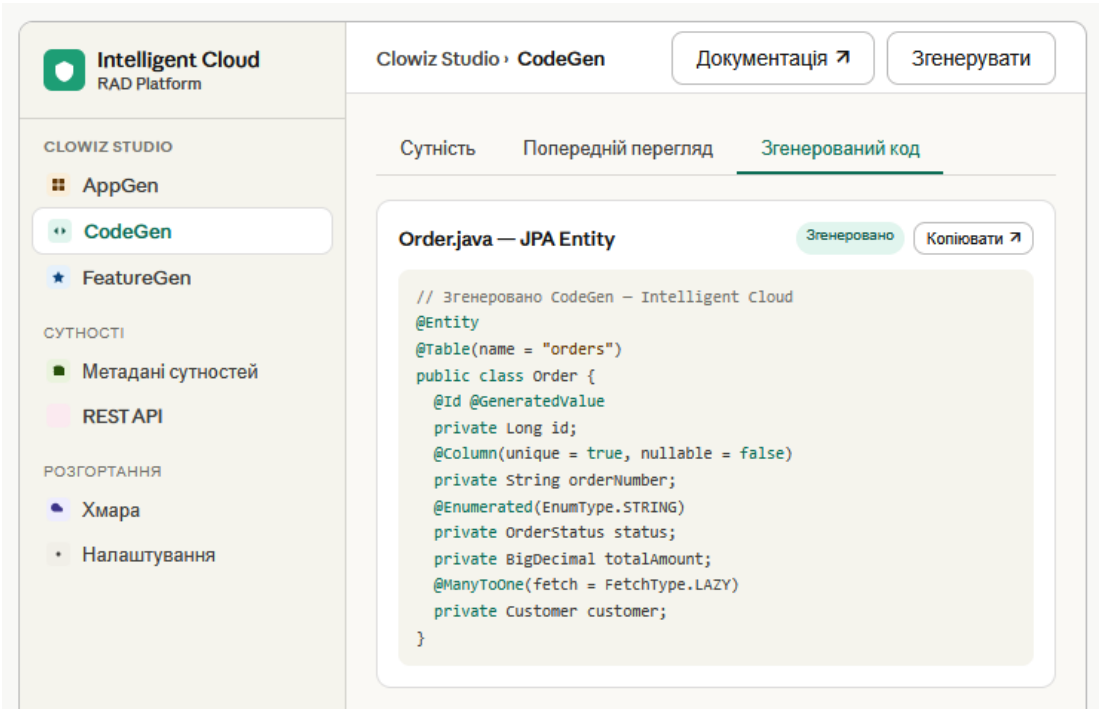


Рисунок 4.17 - Модуль CodeGen (згенерований код)

На рисунку 4.18 подано інтерфейс модуля FeatureGen у межах платформи Intelligent Cloud. Це інструмент високорівневої генерації, який дозволяє створювати не просто окремі класи, а цілісні функціональні блоки (фічі) для бізнес-додатків.

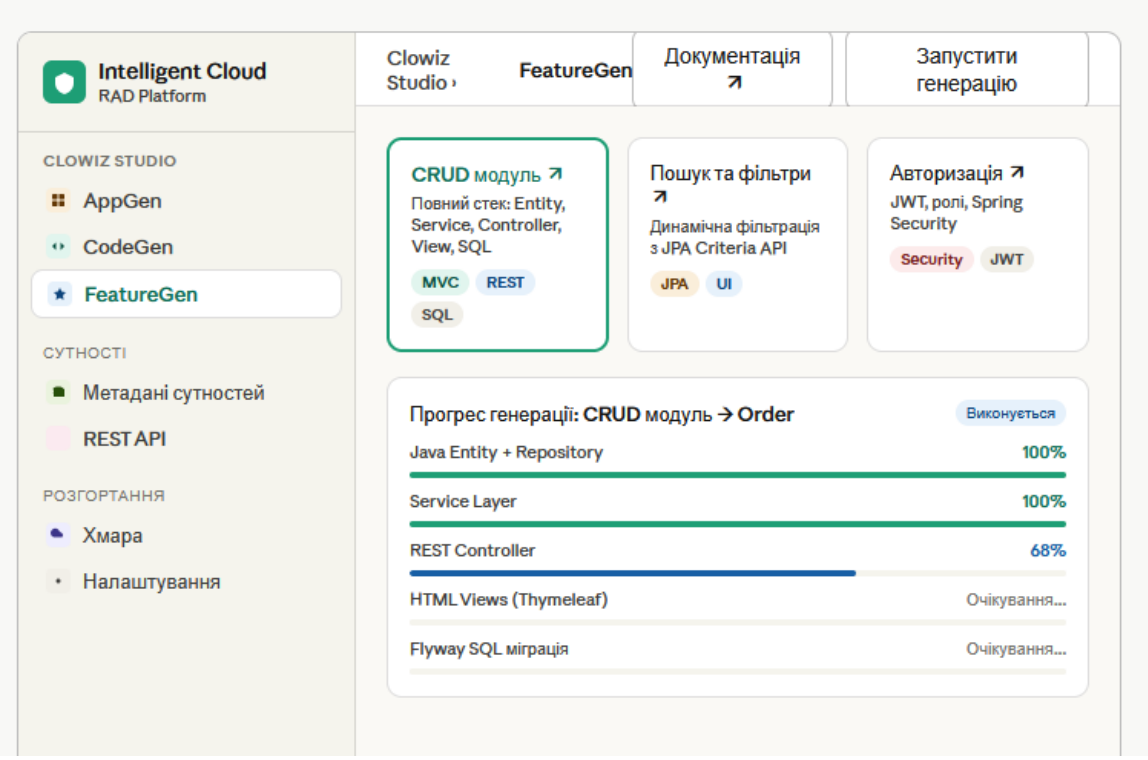


Рисунок 4.18 - Інтерфейс модуля FeatureGen

У верхній частині робочої області розташовані картки доступних «фіч», які можна додати до проєкту:

- CRUD модуль: генерація повного стека для роботи з даними (Entity, Service, Controller, UI-форми та SQL). Підтримує архітектури MVC та REST.

- Пошук та фільтри: інструмент для створення динамічної фільтрації на базі JPA Criteria API.

- Авторизація: модуль безпеки, що інтегрує Spring Security, ролі користувачів та JWT-токени.

У правому верхньому куті розташована кнопка «Запустити генерацію», яка ініціює процес створення коду за обраними параметрами.

Нижня частина інтерфейсу візуалізує поточний статус створення модуля для сутності Order (Замовлення). Ми бачимо стан виконання у реальному часі:

- Статус: «Виконується» (In progress).

- Завершені етапи (100%): рівень даних (Java Entity + Repository) та сервісний шар (Service Layer) вже повністю згенеровані.

- Поточний етап (68%): система зараз працює над створенням REST контролера.

- Черга: генерація інтерфейсів (Thymeleaf) та SQL-міграцій Flyway перебуває у стані очікування.

Цей інтерфейс фокусується на «модульній» збірці застосунку. Замість того, щоб писати кожен контролер вручну, розробник просто обирає необхідну функціональність, а платформа забезпечує консистентність усіх шарів — від бази даних до інтерфейсу користувача.

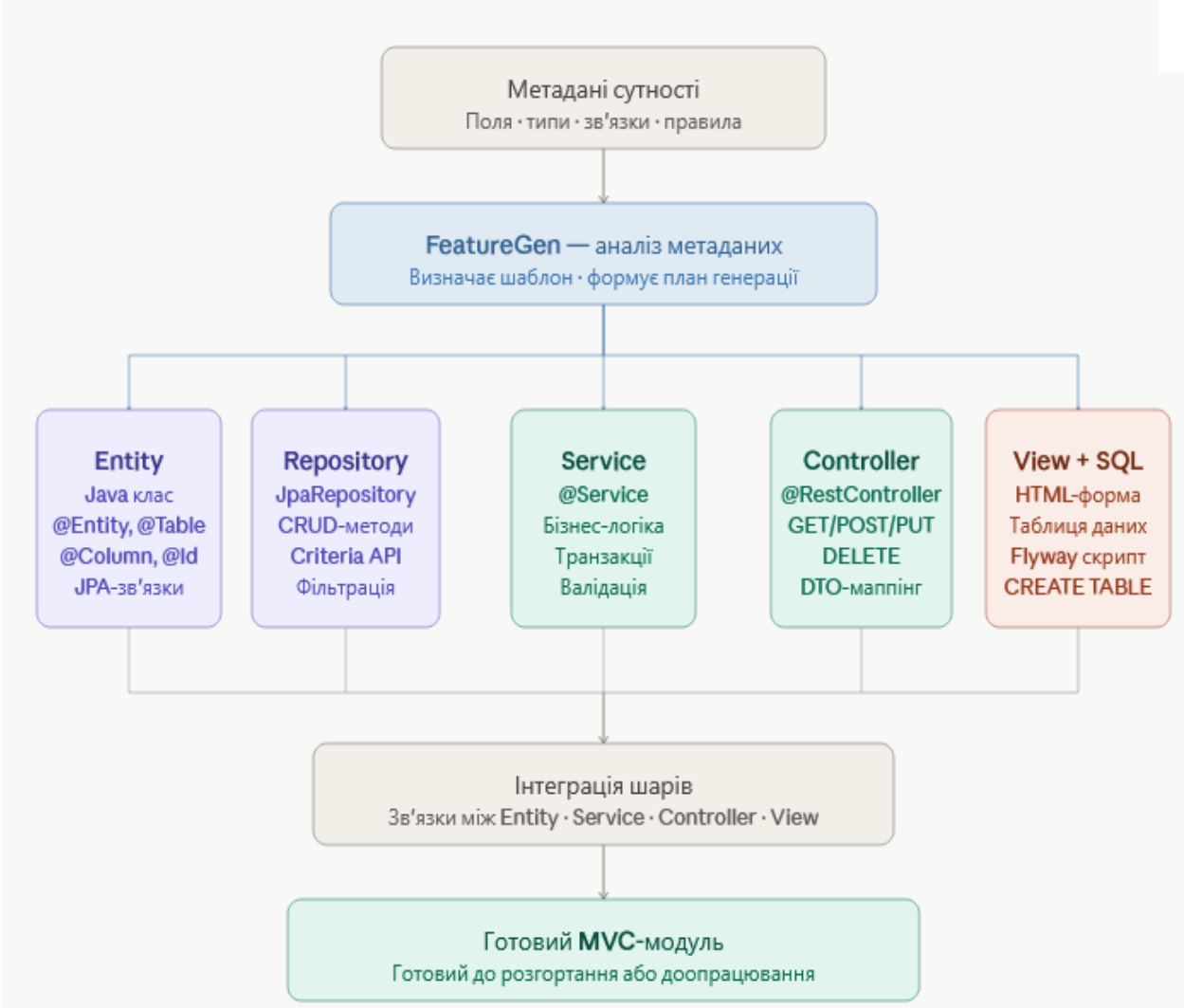


Рисунок 4.19 – Алгоритм роботи на рівні FeatureGen

FeatureGen — це середній рівень платформи між атомарним CodeGen та повним AppGen. Він приймає метадані однієї сутності та генерує цілісний

MVC-модуль за один прохід, де всі шари — Entity, Service, Controller, View і SQL-міграція — вже інтегровані між собою. Ось як виглядає цей процес крок за кроком (рис. 4.19).

Процес складається з чотирьох фаз:

1. Вхід. Користувач описує сутність через метадані — назви полів, типи даних, зв'язки (FK, N:M) та правила валідації. Це єдине, що потрібно визначити вручну.

2. Аналіз. FeatureGen читає метадані, вибирає архітектурний шаблон (MVC-моноліт або мікросервіс) і формує внутрішній план генерації — по суті, список того, що потрібно створити і як ці частини мають посилатися одна на одну.

3. Паралельна генерація п'яти артефактів. Це головна відмінність від CodeGen, який генерує по одному файлу. FeatureGen одночасно створює:

- Entity — JPA-клас з анотаціями @Entity, @Column, зв'язками через @ManyToOne тощо

- Repository — інтерфейс JpaRepository з CRUD-методами та підтримкою Criteria API для фільтрації

- Service — @Service-клас з транзакціями та бізнес-логікою

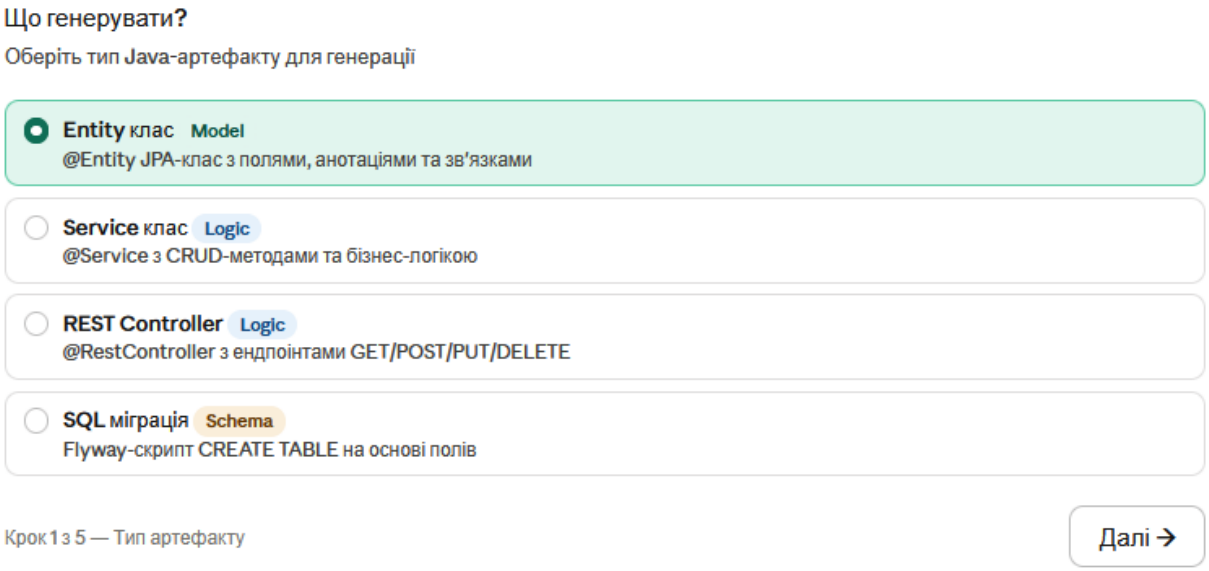
- Controller — @RestController з усіма ендпоінтами GET/POST/PUT/DELETE та DTO-маппінгом

- View + SQL — HTML-форму, таблицю даних і Flyway-скрипт CREATE TABLE

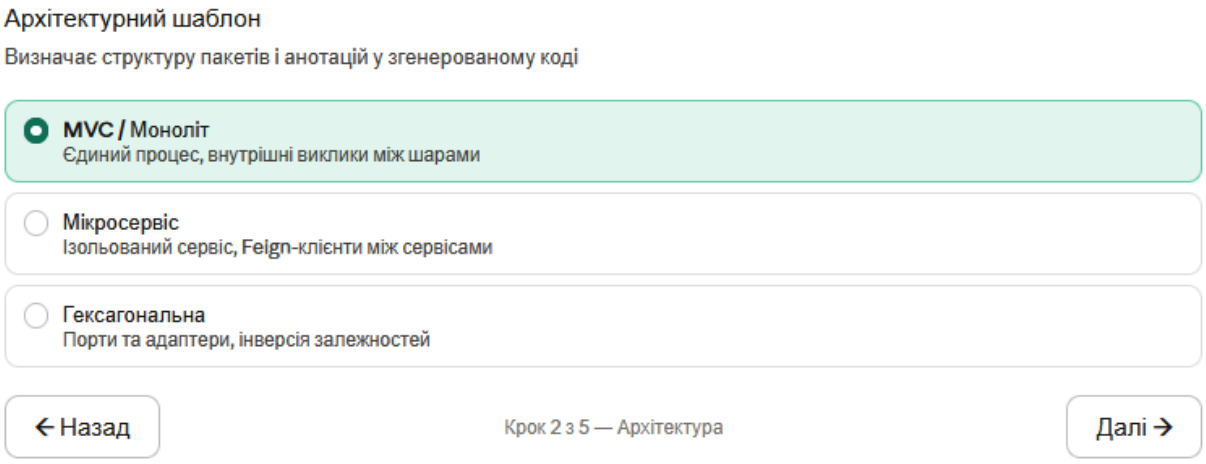
4. Інтеграція. Після генерації всі п'ять артефактів "зшиваються": Controller знає про Service, Service — про Repository, Repository — про Entity, View отримує правильні URL до Controller. Результат — повністю готовий до використання модуль, де між шарами вже немає "ручних" з'єднань.

CodeGen — це інструмент атомарної генерації у платформі. На відміну від FeatureGen, він генерує один конкретний артефакт за раз: Java-клас, HTML-сторінку або SQL-скрипт. Це дає точний контроль над кожним файлом.

На рисунку 4.20 показано етапи роботи інструменту CodeGen.

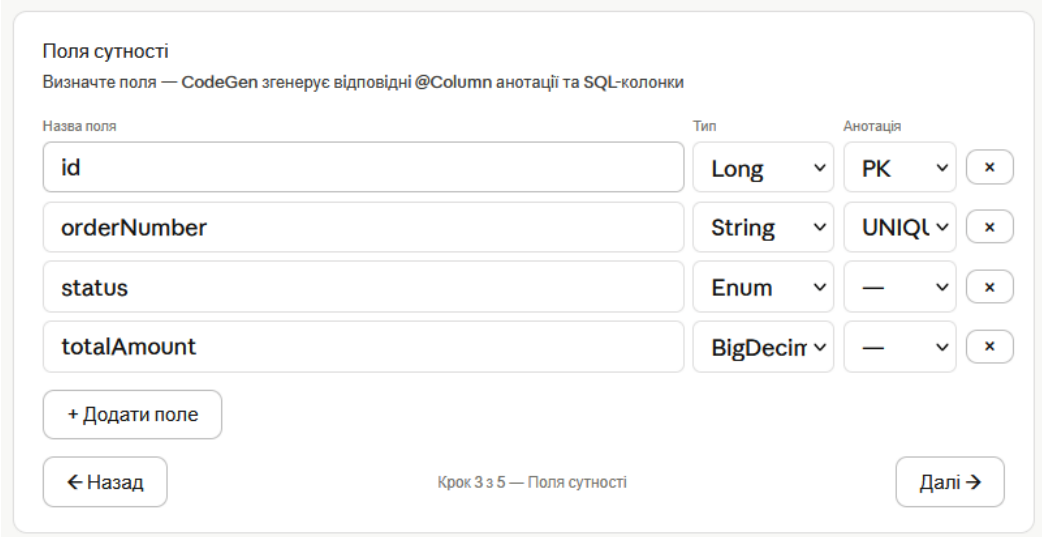


а) крок 1

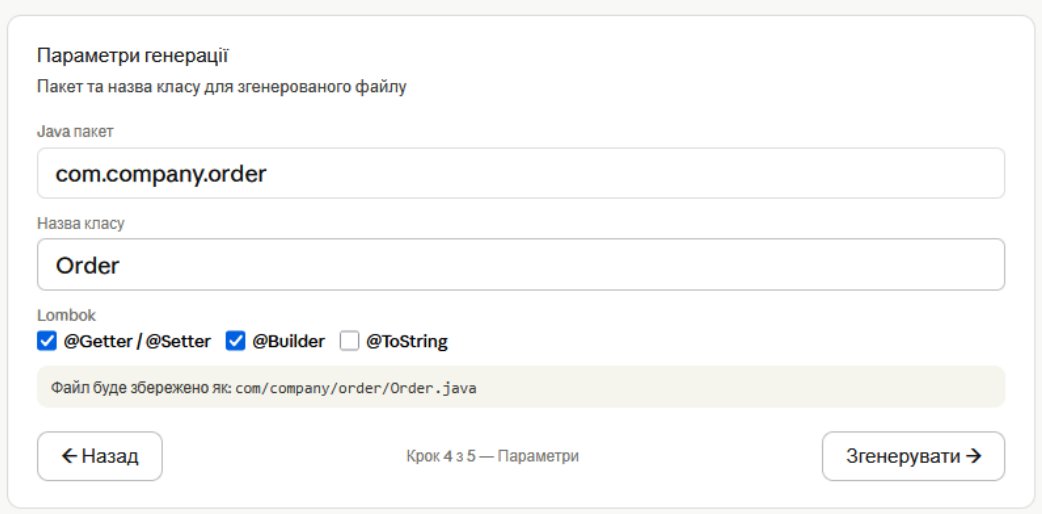


б) крок 2

Рисунок 4.20 - Етапи роботи інструменту CodeGen



в) крок 3



г) крок 4

д) крок 5

Рисунок 4.20 - Етапи роботи інструменту CodeGen

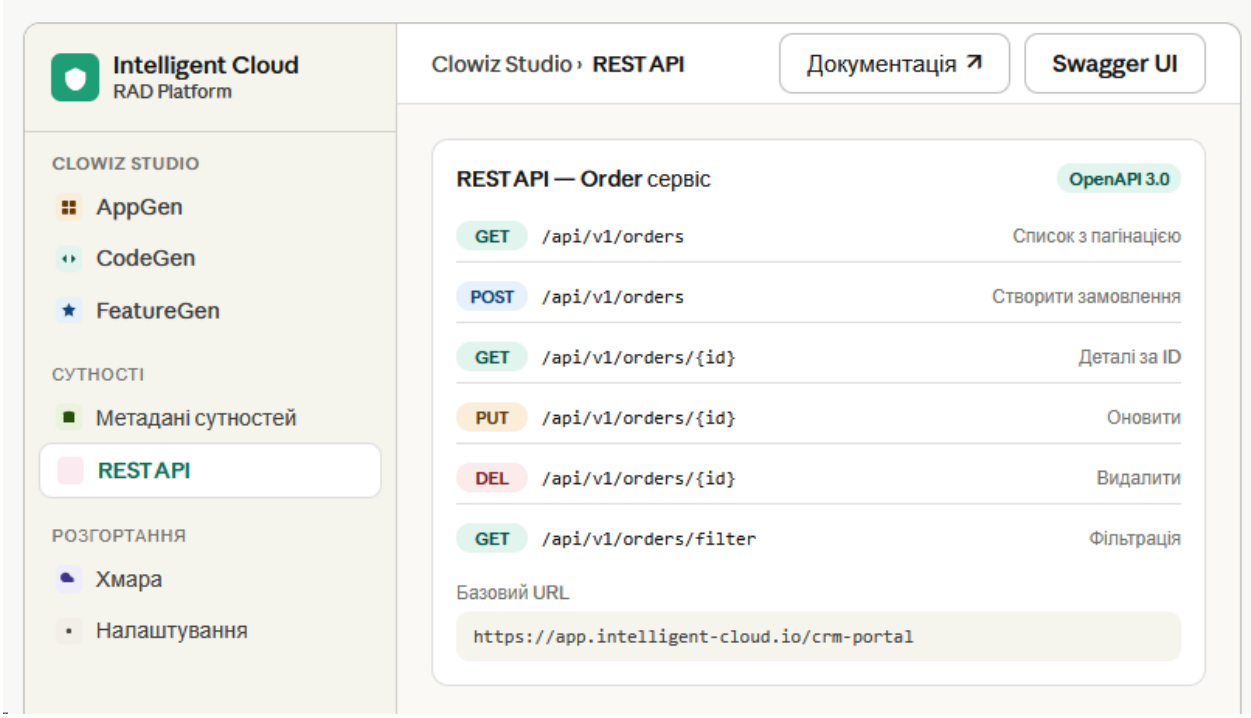


Рисунок 4.21 - Інтерфейс автоматично згенерованої документації REST API

Рисунок 4.21 демонструє інтерфейс автоматично згенерованої документації REST API для конкретного сервісу в межах платформи Intelligent Cloud. Ця панель є частиною інструментарію і призначена для візуалізації, тестування та інтеграції програмних інтерфейсів розробленого застосунку.

Центральна область містить перелік кінцевих точок (endpoints) для сутності «Замовлення» (Order), що відповідають стандарту OpenAPI 3.0:

Методи та шляхи:

- GET /api/v1/orders: отримання повного списку замовлень із підтримкою пагінації.

- POST /api/v1/orders: створення нового запису замовлення в системі.

- GET /api/v1/orders/{id}: запит детальної інформації про конкретне замовлення за його ідентифікатором.

- PUT /api/v1/orders/{id}: модифікація (оновлення) існуючих даних замовлення.

- DEL /api/v1/orders/{id}: остаточне видалення запису з бази даних.

- GET /api/v1/orders/filter: функціонал для складного пошуку та фільтрації замовлень за параметрами.

У верхній правій частині робочої області розташовані елементи для поглибленої роботи з API:

- Документація: перехід до розширеного опису моделей даних та параметрів запитів.

- Swagger UI: кнопка для відкриття інтерактивної пісочниці, де розробники можуть виконувати тестові запити до API у реальному часі.

У нижній частині панелі вказано Базовий URL (https://app.intelligent-cloud.io/crm-portal), який слугує кореневою адресою для всіх API-запитів у межах розгорнутого порталу CRM.

Даний інтерфейс забезпечує повну прозорість взаємодії з бекенд-частиною застосунку. Завдяки автоматичній генерації такої документації, розробники можуть миттєво приступати до інтеграції фронтенду або зовнішніх систем, маючи перед очима актуальний перелік методів та актуальну адресу сервісу.

4.5. Висновки по розділу

У четвертому розділі виконано деталізоване проєктування архітектури фреймворку та визначено структуру його основних компонентів. Розроблено функціонально-логічну модель системи, яка відображає взаємодію елементів фреймворку та забезпечує підтримку повного життєвого циклу хмарних застосунків. Запропонована методологія проєктування на основі метаданих дозволяє підвищити рівень абстракції та автоматизувати ключові етапи розробки.. Реалізована структура та інтерфейс фреймворку забезпечують зручність використання та ефективну взаємодію користувача з платформою.

ВИСНОВКИ

В бакалаврській роботі виконано розроблення платформи-фреймворку для створення хмарних застосунків. У першому розділі проведено ґрунтовний аналіз еволюції архітектурних парадигм, що дозволило встановити закономірності переходу від монолітних систем до сервіс-орієнтованих і мікросервісних архітектур. Доведено, що основними обмеженнями монолітного підходу є низька масштабованість, складність супроводу та обмежена гнучкість у процесі розвитку систем. Водночас мікросервісна архітектура, попри свої переваги, супроводжується підвищеною складністю управління та необхідністю застосування спеціалізованих інструментів оркестрації. Окрему увагу приділено концепції хмарно-орієнтованих застосунків, які розглядаються як логічне продовження розвитку розподілених систем, орієнтованих на динамічне масштабування та ефективне використання обчислювальних ресурсів.

У другому розділі сформовано методологічний базис проєктування програмних систем, у межах якого обґрунтовано роль архітектурних рішень як ключового чинника забезпечення якості програмного продукту. Встановлено, що сучасні підходи до проєктування повинні враховувати не лише функціональні вимоги, але й нефункціональні характеристики, такі як продуктивність, масштабованість, надійність та безпека. Розглянута методологія лінійок програмних продуктів (Software Product Lines) продемонструвала ефективність повторного використання програмних компонентів як засобу підвищення продуктивності розробки та зниження витрат.

Порівняльний аналіз архітектурних парадигм дозволив визначити оптимальні підходи до побудови хмарних застосунків, серед яких ключову роль відіграють модульність, слабка зв’язаність компонентів та використання декларативних моделей конфігурації. У результаті було сформовано систему

критеріїв та атрибутів якості хмарно-орієнтованих систем, що включає адаптивність, відмовостійкість, масштабованість та інтероперабельність.

У третьому розділі здійснено проєктування та інженерну реалізацію фреймворку, що отримав назву Intelligent Cloud. Розроблено його концептуальну модель та визначено архітектурну організацію, яка базується на принципах модульності, розширюваності та автоматизації. Важливим компонентом фреймворку є використання ORM-технологій, що дозволяють автоматизувати процес взаємодії з базами даних та генерацію відповідних скриптів. Це забезпечує підвищення продуктивності розробки та зменшення ймовірності помилок.

У четвертому розділі деталізовано архітектуру фреймворку, описано його основні компоненти та механізми взаємодії між ними. Розроблено функціонально-логічну архітектуру та визначено життєвий цикл хмарних застосунків, що створюються за допомогою фреймворку. Особливу увагу приділено методології проєктування на основі метаданих, яка дозволяє забезпечити високий рівень абстракції та автоматизації процесів розробки.

Запропонована модель станів та життєвого циклу додатка забезпечує контроль над процесами розгортання, експлуатації та масштабування системи. Реалізація механізмів персистентності та інваріантності рівня збереження даних дозволяє гарантувати стабільність роботи застосунків незалежно від змін у фізичній інфраструктурі.

У результаті виконаної роботи розроблено структуру фреймворку та реалізовано його інтерфейс, що забезпечує зручну взаємодію користувача з системою та підтримує основні сценарії розробки хмарних застосунків.

ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Armbrust M., Fox A., Griffith R., Joseph A. D., Katz R., Konwinski A., Lee G., Patterson D., Rabkin A., Stoica I., Zaharia M. A View of Cloud Computing. Communications of the ACM. New York: ACM, 2010, Vol. 53, No. 4, pp. 50–58.

2. Mell P., Grance T. The NIST Definition of Cloud Computing. NIST Special Publication 800-145. Gaithersburg: National Institute of Standards and Technology, 2011, pp. 1–7.

3. Newman S. Microservices: A Definition of This New Architectural Term. IEEE Software. Los Alamitos: IEEE Computer Society, 2015, Vol. 32, No. 5, pp. 8–11.

4. Fowler M., Lewis J. Microservices: A Definition of This Architectural Style. Proceedings of the 12th International Conference on Agile Software Development. Cham: Springer, 2014, pp. 1–6.

5. Dragoni N., Giallorenzo S., Lafuente A. L., Mazzara M., Montesi F., Mustafin R., Safina L. Microservices: Yesterday, Today, and Tomorrow. Present and Ulterior Software Engineering. Cham: Springer, 2017, pp. 195–216.

6. Pahl C. Containerization and the PaaS Cloud. IEEE Cloud Computing. Los Alamitos: IEEE, 2015, Vol. 2, No. 3, pp. 24–31.

7. Merkel D. Docker: Lightweight Linux Containers for Consistent Development and Deployment. Linux Journal. Houston: Belltown Media, 2014, No. 239, pp. 2–6.

8. Burns B., Grant B., Oppenheimer D., Brewer E., Wilkes J. Borg, Omega, and Kubernetes. Communications of the ACM. New York: ACM, 2016, Vol. 59, No. 5, pp. 50–57.

9. Hightower K., Burns B., Beda J. Kubernetes: Up and Running. Proceedings of the USENIX Annual Technical Conference. Berkeley: USENIX Association, 2017, pp. 1–10.

10. Pautasso C., Zimmermann O., Leymann F. RESTful Web Services vs. Big Web Services. Proceedings of the 17th International World Wide Web Conference. New York: ACM, 2008, pp. 805–814.

11. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures. Doctoral Dissertation. Irvine: University of California, 2000, pp. 1–162.

12. Bass L., Clements P., Kazman R. Software Architecture in Practice. Addison-Wesley Professional. Boston: Pearson, 2012, pp. 1–560.

13. Richards M. Software Architecture Patterns. O'Reilly Media Conference Series. Sebastopol: O'Reilly Media, 2015, pp. 1–45.

14. Erl T., Puttini R., Mahmood Z. Cloud Computing: Concepts, Technology & Architecture. Prentice Hall. Upper Saddle River: Pearson, 2013, pp. 1–528.

15. Zhang Q., Chen M., Li L., Li L. Cloud Computing: State-of-the-art and Research Challenges. Journal of Internet Services and Applications. London: Springer, 2010, Vol. 1, No. 1, pp. 7–18.

16. Buyya R., Yeo C. S., Venugopal S., Broberg J., Brandic I. Cloud Computing and Emerging IT Platforms. Future Generation Computer Systems. Amsterdam: Elsevier, 2009, Vol. 25, No. 6, pp. 599–616.

17. Humble J., Farley D. Continuous Delivery. Addison-Wesley. Boston: Pearson, 2011, pp. 1–512.

18. Evans E. Domain-Driven Design. Addison-Wesley. Boston: Pearson, 2004, pp. 1–560.

19. Garlan D., Shaw M. An Introduction to Software Architecture. Advances in Software Engineering and Knowledge Engineering. Singapore: World Scientific, 1993, pp. 1–39.

20. Kruchten P. The 4+1 View Model of Architecture. IEEE Software. Los Alamitos: IEEE, 1995, Vol. 12, No. 6, pp. 42–50.

21. Lewis J., Fowler M. Microservices: A Definition of This Architectural Style. Martin Fowler Technical Reports. Boston: ThoughtWorks, 2014, pp. 1–10.

22. Jamshidi P., Pahl C., Mendonça N. C. Microservices: The Journey So Far and Challenges Ahead. IEEE Software. Los Alamitos: IEEE, 2018, Vol. 35, No. 3, pp. 24–35.

23. Taibi D., Lenarduzzi V., Pahl C. Architectural Patterns for Microservices. Proceedings of the 8th International Conference on Cloud Computing and Services Science. Setúbal: SCITEPRESS, 2018, pp. 221–232.

24. Villamizar M., Garcés O., Ochoa L., Castro H., Salamanca L., Verano M., Casallas R. Infrastructure Cost Comparison of Running Web Applications. Proceedings of the 10th IEEE Cloud Computing Conference. New York: IEEE, 2015, pp. 179–182.

25. Zaharia M., Chowdhury M., Das T., Dave A., Ma J., McCauley M., Franklin M., Shenker S., Stoica I. Resilient Distributed Datasets. Proceedings of the 9th USENIX Conference on Networked Systems Design. Berkeley: USENIX, 2012, pp. 1–14.

26. Dean J., Ghemawat S. MapReduce: Simplified Data Processing. Communications of the ACM. New York: ACM, 2008, Vol. 51, No. 1, pp. 107–113.

27. Bernstein D. Containers and Cloud. IEEE Cloud Computing. Los Alamitos: IEEE, 2014, Vol. 1, No. 3, pp. 81–84.

28. Chen L. Continuous Delivery: Overcoming Adoption Challenges. Journal of Systems and Software. Amsterdam: Elsevier, 2017, Vol. 128, pp. 72–86.

29. Pressman R. S., Maxim B. R. Software Engineering: A Practitioner's Approach. McGraw-Hill. New York: McGraw-Hill, 2014, pp. 1–976.

30. Papazoglou M. P. Service-Oriented Computing. Communications of the ACM. New York: ACM, 2003, Vol. 46, No. 10, pp. 25–28.

31. Josuttis N. SOA in Practice. O'Reilly Media. Sebastopol: O'Reilly, 2007, pp. 1–600.

32. Fowler M. Patterns of Enterprise Application Architecture. Addison-Wesley. Boston: Pearson, 2002, pp. 1–560.

33. Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M. Pattern-Oriented Software Architecture. Wiley. Chichester: Wiley, 1996, pp. 1–457.

БІБЛІОГРАФІЧНА ДОВІДКА

Тема бакалаврської роботи: “ Розроблення платформи - фреймворку для створення хмарних застосунків ”

Обсяг пояснювальної записки: 84 аркуші.

Дата закінчення роботи: 9 червня 2026 р.

Підпис студента _____