Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп’ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)
Кафедра комп’ютерних наук
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

магістр
(назва освітнього ступеня)

на тему: Оптимізація адаптивно-гібридної архітектури ARX для підвищення масштабованості та продуктивності вебклієнтів.

Виконав: студент VI курсу, групи СНнм-61
спеціальності 122 Комп’ютерні науки
(шифр і назва спеціальності)

______ Старицький О.Р.
(підпис) (прізвище та ініціали)

Керівник ______ Никитюк В.В.
(підпис) (прізвище та ініціали)

Нормоконтроль ______ Дуда О.М.
(підпис) (прізвище та ініціали)

Завідувач кафедри ______ Боднарчук І.О.
(підпис) (прізвище та ініціали)

Рецензент ______ Загородна Н.В.
(підпис) (прізвище та ініціали)

Тернопіль
2026

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп’ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)

Кафедра комп’ютерних наук
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
________ Боднарчук І.О.
(підпис) (прізвище та ініціали)
«____» ___________ 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня Магістр
(назва освітнього ступеня)

за спеціальністю 122 Комп’ютерні науки
(шифр і назва спеціальності)

Студенту Старицького Олександру Романовичу
(прізвище, ім’я, по батькові)

1. Тема роботи Оптимізація адаптивно-гібридної архітектури ARX для підвищення масштабованості та продуктивності вебклієнтів.

Керівник роботи Никитюк Вячеслав Вячеславович к.т.н., доцент., доцент кафедри КН
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від « 10 » березня 2026 року № 4/9-150

2. Термін подання студентом завершеної роботи 14.05.26

3. Вихідні дані до роботи Наукові публікації з фронтенд-архітектур і вебпродуктивності, а також експериментальні дані, отримані за допомогою RUM, Lighthouse і Puppeteer на застосунках React/Next.js, використані для оцінювання ефективності архітектури ARX.

4. Зміст роботи (перелік питань, які потрібно розробити)
Вступ. 1 аналіз сучасних вимог до архітектур вебклієнтів та формування методичних основ для дослідження. 2 Порівняльний аналіз існуючих архітектурних моделей фронтенду та передумови створення методології ARX. 3 Оптимізація та практична реалізація архітектурної методології ARX. 4 Експериментальне дослідження ефективності arx та обґрунтування її переваг.
4 Охорона праці та безпека в надзвичайних ситуаціях. Висновки. Додатки

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень, слайдів)
1 Титульна сторінка. 2 Тема, Мета, Об’єкт, Предмет дослідження. 3 Завдання дослідження.
4 Актуальність дослідження. 5. Дослідження фронтенд-архітектур та адаптивного рендерингу.
6. Обґрунтування вибору архітектури ARX. 7. Порівняння архітектурних підходів фронтенду.
8. Метрики та інструменти оцінювання ARX. 9. Реалізація та інтеграція ARX у React/Next.js.
10. Інтеграція ARX із AI-системами, організація коду та автоматизація розробки.
11. Аналіз продуктивності, масштабованості та переваг ARX у порівнянні з традиційними архітектурами. 12 Висновки.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Охорона праці	Сенчишин В.С., доцент каф. МТ		
Безпека в надзвичайних ситуаціях	Теслюк В.М., проректор з адміністративно-господарської роботи та будівництва		

7. Дата видачі завдання 10 березня 2026 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	Ознайомлення з завданням до кваліфікаційної роботи	13.03.2026	*Виконано*
2.	Підбір та опрацювання наукових публікацій, збір даних по темі роботи	16.03.2026-20.03.2026	*Виконано*
3.	Виконання дослідження згідно теми кваліфікаційної роботи	21.03.2026-29.03.2026	*Виконано*
4.	Оформлення розділу «Аналіз сучасних вимог до архітектур вебклієнтів та формування методичних основ для дослідження»	29.03.2026-03.04.2026	*Виконано*
5.	Оформлення розділу «Порівняльний аналіз існуючих архітектурних моделей фронтенду та передумови створення методології ARX»	06.04.2026-17.04.2026	*Виконано*
6.	Оформлення розділу «Оптимізація та практична реалізація архітектурної методології ARX»	26.04.2026-17.05.2026	*Виконано*
7.	Оформлення розділу «Експериментальне дослідження ефективності arx та обґрунтування її переваг»	20.05.2026-24.05.2026	*Виконано*
8.	Виконання завдання до підрозділу «Охорона праці»	27.04.2026-10.05.2026	*Виконано*
9.	Виконання завдання до підрозділу «Безпека в надзвичайних ситуаціях»	27.04.2026-10.05.2026	*Виконано*
ґ0.	Оформлення кваліфікаційної роботи	04.05.2026-08.05.2026	*Виконано*
ґґ.	Нормоконтроль	06.05.2026	*Виконано*
11.	Перевірка на плагіат	07.05.2026	*Виконано*
12.	Попередній захист кваліфікаційної роботи	21.05.2026	*Виконано*
13.	Захист кваліфікаційної роботи	28.05.2026	

Студент ____________ Старицький О.Р.
(підпис) (прізвище та ініціали)

Керівник роботи ____________ Никитюк В.В.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Тема: Оптимізація адаптивно-гібридної архітектури ARX для підвищення масштабованості та продуктивності вебклієнтів. // Кваліфікаційна робота освітнього рівня «Магістр» // Старицький Олександр Романович// Тернопільський національний технічний університет імені Івана Пулюя, факультет комп'ютерно-інформаційних систем і програмної інженерії, кафедра комп'ютерних наук, група СНнм-61 // Тернопіль, 2026 // С. 86, рис. – 14, додат. – 4, бібліогр. – 71.

Ключові слова: веброзробка, фронтенд-архітектура, JavaScript фреймворки, React.js, Next.js, продуктивність, оптимізація, масштабованість, SEO.

У роботі проведено дослідження та формалізацію адаптивно-гібридної архітектури ARX, спрямованої на підвищення продуктивності, масштабованості та ефективності сучасних вебзастосунків. Проаналізовано сучасні фронтенд-архітектури (SPA, SSR, SSG, Islands) та їхні обмеження, на основі чого сформовано систему критеріїв оцінювання (Performance, Developer Experience, Maintainability, Scalability) і запропоновано методологію ARX, що поєднує гібридний рендеринг, контекстно-адаптивне прийняття рішень і оркестрацію інтерфейсу. Ефективність підходу підтверджено експериментально за допомогою RUM, Lighthouse та Puppeteer на тестових застосунках React/Next.js, де ARX продемонструвала покращення ключових метрик продуктивності (FCP, LCP, TTI, TBT), зменшення складності реалізації та підвищення швидкості розробки й масштабованості системи.

ANNOTATION

Title: Optimization of the adaptive-hybrid ARX architecture to improve scalability and performance of web clients // Master's qualification thesis // Oleksandr Starytskyi // Ternopil Ivan Puluj National Technical University, Faculty of Computer Information Systems and Software Engineering, Department of Computer Science, group СНм-61 // Ternopil, 2026 // p. 86, figures – 14, appendices – 4, references – 71.

Keywords: web development, frontend architecture, JavaScript frameworks, React.js, Next.js, performance, optimization, scalability, SEO.

The thesis presents a study and formalization of the adaptive-hybrid ARX architecture aimed at improving the performance, scalability, and efficiency of modern web applications. Modern frontend architectures (SPA, SSR, SSG, Islands) and their limitations are analyzed, which serves as a basis for defining an evaluation framework (Performance, Developer Experience, Maintainability, Scalability) and proposing the ARX methodology that combines hybrid rendering, context-aware decision-making, and interface orchestration. The effectiveness of the approach is experimentally validated using RUM, Lighthouse, and Puppeteer on test applications built with React/Next.js, where ARX demonstrates improvements in key performance metrics (FCP, LCP, TTI, TBT), reduced implementation complexity, and enhanced development speed and system scalability.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

AI (англ. Artificial Intelligence) – штучний інтелект.

ARX (англ. Adaptive Rendering eXperience) – адаптивно-гібридна архітектура рендерингу.

CLS (англ. Cumulative Layout Shift) – метрика зсуву елементів сторінки.

CSR (англ. Client-Side Rendering) – клієнтський рендеринг.

DX (англ. Developer Experience) – досвід розробника.

FCP (англ. First Contentful Paint) – час до першого відображення контенту.

LCP (англ. Largest Contentful Paint) – час до відображення найбільшого елемента сторінки.

LOC (англ. Lines of Code) – кількість рядків коду.

RUM (англ. Real User Monitoring) – моніторинг реальних користувачів.

SPA (англ. Single Page Application) – односторінковий вебзастосунок.

SSG (англ. Static Site Generation) – статична генерація сторінок.

SSR (англ. Server-Side Rendering) – серверний рендеринг.

TBT (англ. Total Blocking Time) – загальний час блокування основного потоку.

TTI (англ. Time to Interactive) – час до повної інтерактивності сторінки.

API (англ. Application Programming Interface) – інтерфейс програмування застосунків.

CI/CD (англ. Continuous Integration / Continuous Deployment) – безперервна інтеграція та розгортання.

DOM (англ. Document Object Model) – об’єктна модель документа.

SEO (англ. Search Engine Optimization) – оптимізація для пошукових систем.

UI (англ. User Interface) – інтерфейс користувача.

UX (англ. User Experience) – користувацький досвід.

ЗМІСТ

ВСТУП **9**

1 АНАЛІЗ СУЧАСНИХ ВИМОГ ДО АРХІТЕКТУР ВЕБКЛІЄНТІВ ТА ФОРМУВАННЯ МЕТОДИЧНИХ ОСНОВ ДЛЯ ДОСЛІДЖЕННЯ **11**

1.1 Сучасні тенденції у фронтенд-розробці 11

1.2 Основні критерії вибору архітектури: продуктивність, підтримуваність, масштабованість, розробницький досвід 13

1.2.1 Архітектурне рішення у фронтенд-розробці 13

1.2.2 Продуктивність та підтримуваність 14

1.2.3 Масштабованість 16

1.4 Визначення системи метрик для оцінки архітектурних рішень 19

1.4.1 Значення системи метрик та їх роль у фронтенд-розробці 19

1.4.2 Метрики продуктивності та розробницького досвіду 20

1.4.3 Метрики підтримуваності та масштабованості 22

1.4.4 Використання системи метрик для оптимізації архітектури 23

1.5 Формування методичних засад подальшого дослідження і постановка завдань для розробки власного підходу 24

1.6 Висновки до першого розділу 26

2. ПОРІВНЯЛЬНИЙ АНАЛІЗ ІСНУЮЧИХ АРХІТЕКТУРНИХ МОДЕЛЕЙ ФРОНТЕНДУ ТА ПЕРЕДУМОВИ СТВОРЕННЯ МЕТОДОЛОГІЇ ARX **28**

2.1. Модель SPA 28

2.1.1 Принцип роботи SPA 28

2.1.2 Продуктивність і UX у SPA 30

2.1.3 Обмеження SEO і шляхи їх вирішення 33

2.2 Методика порівняння архітектур і вибір метрик 35

2.3 Архітектура ARX 39

2.3.1. Концепція та цілі ARX 40

2.3.2. Адаптивний рендеринг у ARX 41

2.3.3. Застосування ARX у IA-фронтенді 42

2.4 Висновки до другого розділу 42

3. ОПТИМІЗАЦІЯ ТА ПРАКТИЧНА РЕАЛІЗАЦІЯ АРХІТЕКТУРНОЇ МЕТОДОЛОГІЇ ARX 44

3.1. Архітектурна концепція ARX: гібридна модель рендерингу та принципи адаптивності 44

3.2. Компоненти системи ARX: рендер-оркестратор, профайлер, build-пайплайн, система політик, AI-генерація scaffold. 46

3.3. Реалізація базового прототипу ARX у середовищі React/Next.js із підтримкою серверних компонентів. 48

3.4. Інтеграція ARX із AI-проєктами: забезпечення сумісності та використання AI для автоматичного тестування 50

3.5. Тестування архітектури на реальних прикладах для перевірки гнучкості, складності та швидкості реалізації елементів 52

3.6 Висновки до третього розділу 54

4. ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ARX ТА ОБҐРУНТУВАННЯ ЇЇ ПЕРЕВАГ 55

4.1 Інструменти аналізу продуктивності ARX 55

4.1.1 Методика оцінювання архітектури ARX: опис використаних метрик та інструментів 55

4.1.2 Оцінювання за допомогою інструменту Lighthouse 56

4.1.3 Використання RUM для аналізу реального користувацького досвіду 59

4.1.4 Оцінювання за допомогою Puppeteer 61

4.2 Оцінка ефективності розробки 63

4.2.1 Аналіз метрик Developer Experience 63

4.2.2 Аналіз кількості коду (LOC) 64

4.3 Оцінка показників Maintainability і Scalability 66

4.4 Порівняння результатів ARX із існуючими архітектурами 67

4.5 Приклади організації коду та розподілу відповідальності між командами у межах ARX 69

4.6 Висновки до четвертого розділу 71

5 ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ 73

5.1 Облаштування і безпека серверних приміщень 73

5.2 Пожежна безпека в навчальних закладах 76

ВИСНОВКИ 79

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ 80

ВСТУП

Актуальність теми. У сучасних умовах стрімкого розвитку вебтехнологій та зростання вимог до продуктивності, масштабованості й адаптивності вебзастосунків особливої актуальності набувають питання оптимізації фронтенд-архітектур. Існуючі підходи, такі як CSR, SSR, SSG та їхні гібридні модифікації, мають низку обмежень, пов'язаних із нерівномірним розподілом обчислювального навантаження, складністю масштабування та зниженням ефективності в умовах динамічних інтерфейсів. Крім того, розвиток AI-орієнтованої розробки вимагає нових архітектурних підходів, здатних забезпечити автоматизацію, адаптивність і ефективне управління рендерингом. У зв'язку з цим розроблення та оптимізація адаптивно-гібридної архітектури ARX, що поєднує різні стратегії рендерингу та забезпечує контекстно-орієнтоване прийняття рішень, є актуальним напрямом наукових досліджень.

Мета і задачі дослідження. Метою даної кваліфікаційної роботи освітнього рівня «Магістр» є підвищення продуктивності та масштабованості вебклієнтів шляхом оптимізації адаптивно-гібридної архітектури ARX. Для досягнення поставленої мети необхідно вирішити такі завдання:

- проаналізувати сучасний стан фронтенд-архітектур та тенденції їх розвитку;
- дослідити існуючі підходи до рендерингу вебзастосунків (CSR, SSR, SSG, Islands) та визначити їх обмеження;
- сформувати систему метрик для оцінювання ефективності архітектурних рішень;
- розробити та формалізувати архітектурну методологію ARX;
- реалізувати прототип системи на базі сучасних технологій (React, Next.js);
- провести експериментальне дослідження ефективності ARX із використанням сучасних інструментів аналізу продуктивності;

- виконати порівняння ARX з існуючими архітектурними підходами.

Об’єкт дослідження – процеси побудови та оптимізації фронтенд-архітектур вебзастосунків.

Предмет дослідження – методи адаптивного управління рендерингом і оптимізації продуктивності вебклієнтів у межах гібридних архітектур.

Наукова новизна одержаних результатів полягає у розробленні та формалізації адаптивно-гібридної архітектури ARX, яка базується на концепції контекстно-орієнтованого вибору стратегії рендерингу та введенні механізму рендер-оркестрації. Запропонований підхід розширює існуючі моделі побудови вебзастосунків шляхом інтеграції адаптивних механізмів управління, що дозволяє розглядати процес рендерингу як керовану динамічну систему.

Практичне значення одержаних результатів. У межах роботи розроблено прототип архітектури ARX, який може бути використаний для побудови сучасних вебзастосунків із підвищеними вимогами до продуктивності та масштабованості. Отримані результати можуть бути застосовані у практиці розробки фронтенд-систем, а також інтегровані в AI-орієнтовані середовища розробки.

Апробація результатів магістерської роботи. Основні результати дослідження обговорювались на IX міжнародної студентської науково-технічної конференції «Природничі та гуманітарні науки. Актальні питання» Тернопільського національного технічного університету імені Івана Пулюя (м. Тернопіль, 2026 р.).

Публікації. Основні результати кваліфікаційної роботи відображені у наукових публікаціях автора. (Див. додаток В1, додаток В2)

Структура й обсяг кваліфікаційної роботи. Кваліфікаційна робота складається зі вступу, чотирьох розділів, висновків, списку використаних джерел та додатків. Загальний обсяг роботи становить 86 сторінок, з них 69 сторінок основного тексту, який містить 14 рисунків та 6 таблиць, додатків 4.

1 АНАЛІЗ СУЧАСНИХ ВИМОГ ДО АРХІТЕКТУР ВЕБКЛІЄНТІВ ТА ФОРМУВАННЯ МЕТОДИЧНИХ ОСНОВ ДЛЯ ДОСЛІДЖЕННЯ

У сучасному світі цифрових технологій фронтенд-застосунки є головним засобом взаємодії клієнта з різними вебсистемами, онлайн-платформами та сервісами. Саме від ретельності вибору їхньої архітектури залежить не тільки швидкість їхньої роботи але також стабільність, масштабованість та зручність використання. З неперервним розвитком сучасних технологій, а саме появу штучного інтелекту - інструмента для створення онлайн платформи, задав стрімкий зростаючий тренд у розробці фронтенд застосунків, які визначають новий напрям трансформації вебзастосунків та відповідно і архітектурних підходів для їх розробки.

1.1 Сучасні тенденції у фронтенд-розробці

Сучасна веброзробка охоплює як статичні, так і динамічні підходи до побудови інтерфейсів. Базовою основою більшості вебсторінок є HTML і CSS, що відповідають за структуру документа та його візуальне оформлення. Такий підхід ефективно застосовується для інформаційних ресурсів, корпоративних сторінок, каталогів і контентних платформ, де вміст не потребує постійного оновлення в реальному часі.

Для реалізації інтерактивної поведінки використовується JavaScript, який дозволяє змінювати інтерфейс без перезавантаження сторінки, обробляти події користувача, виконувати валідацію форм, анімації та працювати з віддаленими даними. Це дає змогу створювати більш адаптивні та функціонально насичені вебзастосунки.

Для спрощення розробки клієнтської частини застосовуються бібліотеки та фреймворки, що стандартизують роботу з DOM, маршрутизацією, станом застосунку та компонентним підходом. Одним із

поширених рішень є архітектура Single Page Application (SPA), де логіка інтерфейсу виконується на стороні браузера, а навігація реалізується без повного перезавантаження сторінки. SPA забезпечує високий рівень інтерактивності та зручний користувацький досвід, але має недоліки: довший первинний завантаження, залежність від JavaScript, складність SEO та підвищене навантаження на клієнтський пристрій [1, 2].

Для вирішення цих обмежень використовуються альтернативні підходи рендерингу, зокрема Server-Side Rendering (SSR) та Static Site Generation (SSG). SSR формує HTML на сервері до відправки сторінки користувачу, що покращує швидкість першого відображення та SEO. SSG передбачає генерацію сторінок під час збірки проєкту, забезпечуючи високу продуктивність і мінімальне навантаження на серверну інфраструктуру [3].

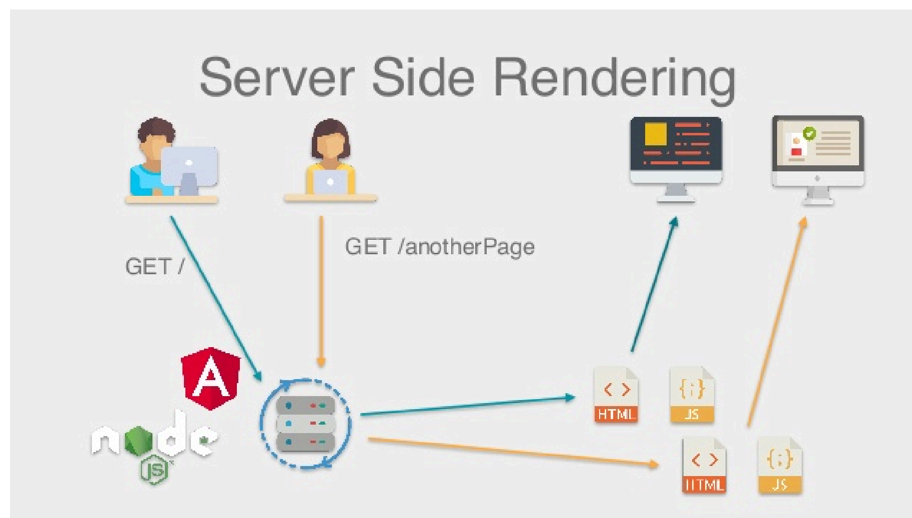


Рис. 1.1.1 - Приклад архітектурного підходу SSR на базі Angular та node.

Серверний рендеринг забезпечує формування HTML на сервері до передачі в браузер, що покращує швидкість первинного відображення та індексацію. Статична генерація сторінок передбачає створення сторінок на етапі збірки з подальшим розміщенням у середовищі доставки контенту, що зменшує навантаження на сервер і забезпечує швидку віддачу сторінок. Обидва підходи ефективні для ресурсів із великою кількістю статичного або напівдинамічного контенту [4].

Фреймворки на кшталт Next.js, Nuxt та Remix підтримують гібридні моделі рендерингу, поєднуючи SPA та серверне формування сторінок, що покращує швидкість першого відгуку та доступність для пошукових систем.

Мікрофронтенди передбачають розподіл застосунку на незалежні модулі з окремими циклами розробки та технологічними стеками, що підвищує гнучкість і масштабованість, але вимагає узгодженої інтеграції та стандартизації взаємодії між модулями [5].

У сучасних фронтенд-системах інтегруються AI-компоненти для персоналізації інтерфейсу, автоматизації процесів та адаптації до поведінки користувача, що підвищує вимоги до управління станом, кешування та безпеки даних. Додатковими підходами оптимізації виступають монорепозиторії, server components та edge rendering, які відповідно спрощують керування кодовою базою, зменшують обсяг клієнтського коду та скорочують затримки обробки запитів [6].

1.2 Основні критерії вибору архітектури: продуктивність, підтримуваність, масштабованість, розробницький досвід

1.2.1 Архітектурне рішення у фронтенд-розробці

Архітектурне рішення у фронтенд-розробці визначає структуру застосунку, взаємодію компонентів, потік даних, управління станом і логіку рендерингу. Також охоплює інтеграцію із зовнішніми сервісами, масштабованість і супровід проєкту. Архітектура відрізняється від технологічного стеку, оскільки задає принципи взаємодії інструментів і розвиток системи протягом життєвого циклу продукту [7].

Структура компонентів, стан і потік даних впливають на ефективність розробки, тестування та підтримки. Якісна архітектура забезпечує повторне використання коду, стабільність і масштабованість. Логіка рендерингу впливає на продуктивність і користувацький досвід. Невдала організація

архітектури призводить до фрагментації коду, складності підтримки та обмежень масштабування, особливо у великих командах [8].

Основні архітектурні підходи включають MVC, SPA, SSR, SSG та мікрофронтенди. MVC розділяє логіку, представлення та дані. SPA забезпечує інтерактивність і динамічні оновлення, але має обмеження в SEO та початковій продуктивності. SSR і SSG покращують швидкість першого завантаження та SEO, але ускладнюють оновлення даних. Мікрофронтенди дозволяють розділяти інтерфейс на незалежні модулі для автономної роботи команд із різними технологіями [9].

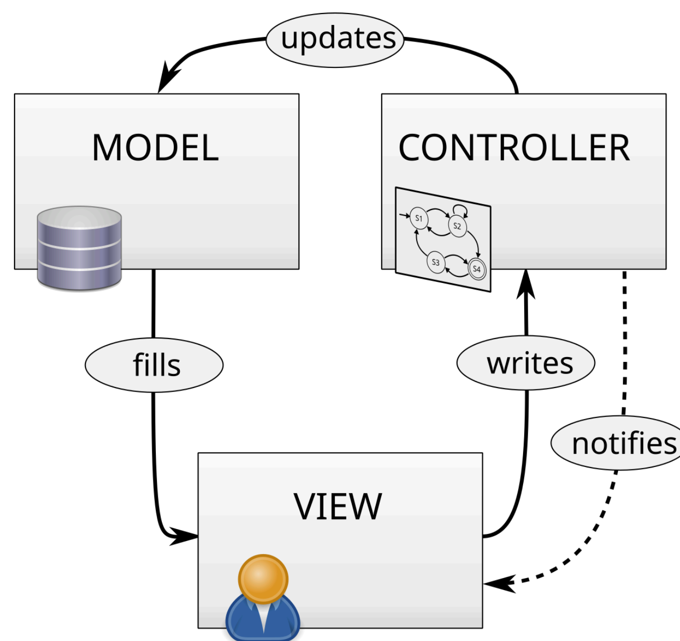


Рис. 1.2.2 - Діаграма MVC

Стратегічне значення архітектурних рішень стає найбільш очевидним у великих довгострокових проектах. Вони визначають не лише якість користувацького досвіду, але й ефективність співпраці, інтеграцію нових функцій та адаптивність до змін у бізнесі. Вибір правильної архітектури на ранній стадії допомагає запобігти технічному боргу, спрощує обслуговування та забезпечує міцну основу для сучасних досягнень, таких як серверні

компоненти, edge rendering та генерація інтерфейсів на основі штучного інтелекту. [10]

1.2.2 Продуктивність та підтримуваність

Продуктивність фронтенду визначає швидкість і плавність взаємодії користувача з вебзастосунком, а також час відображення контенту та реакції інтерфейсу. Вона впливає на користувацький досвід і конверсію, оскільки затримки можуть знижувати залученість. Оцінка продуктивності базується на ключових метриках: TTFB (Time to First Byte), LCP (Largest Contentful Paint), TTI (Time to Interactive), TBT (Total Blocking Time) та FPS (Frames Per Second). Вони відображають швидкість відповіді сервера, час відображення основного контенту, момент повної інтерактивності, блокування основного потоку та плавність інтерфейсу. Сукупно ці метрики характеризують користувацький досвід і дозволяють виявляти проблемні ділянки продуктивності [11].

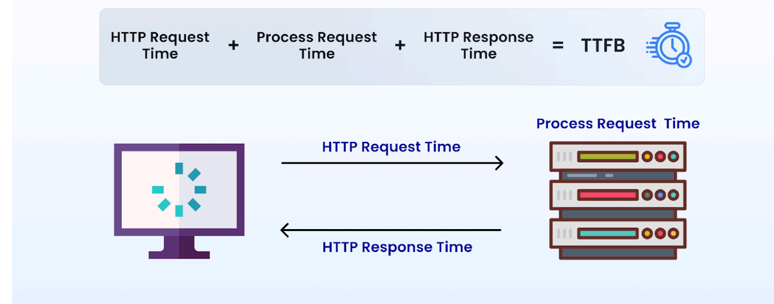


Рис 1.2.1. - Приклад як вираховується TTFB

На швидкодію фронтенду впливають організація потоків даних, кешування, стратегія завантаження ресурсів і баланс між клієнтським та серверним рендерингом. Оптимізація потоків даних зменшує кількість зайвих запитів, кешування скорочує час доступу до даних і навантаження на сервер. Lazy loading зменшує обсяг первинного завантаження. Гібридні підходи SSR і

клієнтського рендерингу оптимізують час першого відображення без втрати інтерактивності [12].

Підтримуваність визначається структурою коду, модульністю, стандартизацією та документацією. Модульна архітектура забезпечує ізоляцію функціональних частин, спрощуючи їх зміну та повторне використання. Єдині стандарти кодування та документація підвищують узгодженість розробки та швидкість адаптації нових учасників [13].

Підвищення підтримуваності забезпечується TypeScript, ESLint, Prettier, принципами SOLID, а також автоматизованим тестуванням і CI/CD-процесами, що підвищують стабільність і передбачуваність змін [14].

Продуктивність і підтримуваність можуть конфліктувати: додаткова абстракція та модульність підвищують підтримуваність, але можуть знижувати продуктивність, тоді як оптимізації для швидкодії ускладнюють супровід коду. Баланс визначається вимогами продукту, навантаженням, розміром команди та бізнес-цілями.

1.2.3 Масштабованість

Масштабованість у фронтенд-архітектурі — це здатність системи ефективно зростати під навантаженням користувачів, функціоналу та команд без втрати продуктивності й стабільності. Вона охоплює як технічні аспекти (обсяг коду, продуктивність), так і організаційні — паралельну роботу команд без конфліктів у кодовій базі [15].

Основним механізмом масштабованості є модульність, що забезпечує розділення системи на незалежні частини з мінімальними взаємозалежностями. Це дозволяє ізолювати логіку, стан і залежності, знижуючи ризики конфліктів і спрощуючи додавання нового функціоналу.

Ізоляція компонентів забезпечує незалежність змін між модулями через контроль інтерфейсів, використання контрактів даних і обмеження доступу до внутрішньої реалізації, що зменшує кількість помилок у великих системах [16].

Мікрофронтенди застосовуються для масштабування великих застосунків через поділ на автономні підсистеми з незалежними командами, технологіями та життєвими циклами. Це зменшує технічний борг і прискорює розробку, за умови дотримання стандартів інтеграції та узгодженого користувацького досвіду [17].

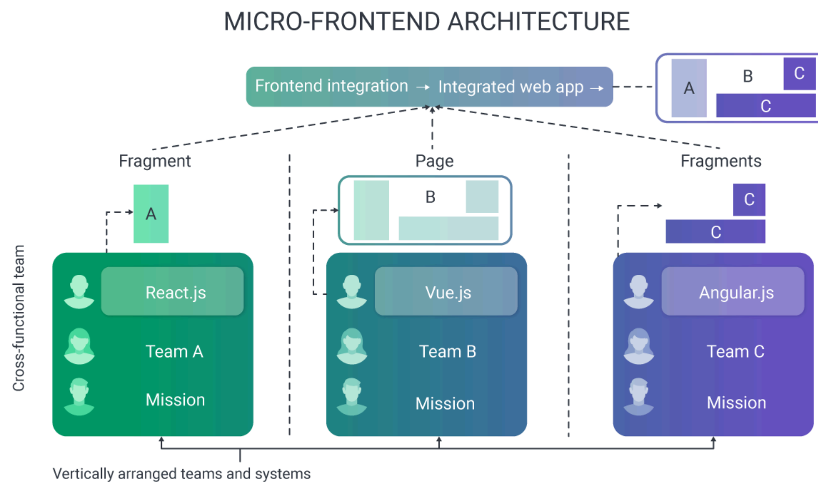


Рис.1.2.1. - Приклад мікрофронтендної архітектури

Без масштабованої архітектури зростає складність коду та залежностей, що ускладнює внесення змін, збільшує кількість помилок і час збірки та тестування. Одночасна робота команд над спільними компонентами призводить до конфліктів, а відсутність модульності — до дублювання функціоналу та складності повторного використання коду. Рішення включають модульну структуру, стандартизовані контракти між компонентами, моніторинг залежностей і CI/CD для незалежного розгортання модулів [18].

Правильна архітектура забезпечує паралельну роботу команд і незалежне розгортання модулів. Команди можуть відповідати за окремі підсистеми або мікрофронтенди без постійної синхронізації, що скорочує цикл доставки змін і підвищує стабільність системи. Масштабованість також дозволяє інтегрувати нові сервіси без суттєвої перебудови існуючої архітектури, забезпечуючи її гнучкість і адаптивність до змін бізнес-вимог.

1.2.4 Досвід розробника та взаємозв’язок критеріїв

Developer Experience охоплює умови середовища розробки, що визначають зручність, швидкість і ефективність роботи з проєктом. Він включає налаштування середовища, інструменти, документацію, CI/CD, дебагінг і тестування. Якісний DX забезпечує швидке включення нових розробників у проєкт і мінімізує час адаптації, що особливо важливо для великих команд [19].

DX впливає на швидкість розробки, якість коду та стабільність системи. Використання сучасних інструментів збірки скорочує час розробки, CI/CD автоматизує тестування, перевірки та деплой, а документація і стандарти кодування спрощують навігацію в проєкті та зменшують технічний борг [20].

DX тісно пов’язаний із продуктивністю, підтримуваністю та масштабованістю. Висока продуктивність може ускладнювати налаштування середовища, масштабовані архітектури вимагають стандартизації, а підходи до підтримуваності можуть збільшувати складність початкового входу в проєкт. У результаті виникає необхідність балансування між цими характеристиками залежно від цілей продукту та ресурсів команди [21].

Розвиток DX підтримується автоматизацією форматування, лінтингу, статичного аналізу, генерації документації, використанням монорепозиторіїв, централізованих CI/CD та інструментів моніторингу продуктивності, що зменшує ручні операції і підвищує стабільність розробки [23].

DX виступає інтегруючим фактором архітектури, забезпечуючи баланс між продуктивністю, підтримуваністю та масштабованістю і впливаючи на швидкість та якість розробки у довгостроковій перспективі.

1.3 Аналіз проблем існуючих підходів при роботі з великими командами та AI-компонентами

Зі зростанням складності фронтенд-застосунків посилюються організаційні та технічні проблеми, пов'язані з узгодженістю архітектури, синхронізацією змін і управлінням залежностями. У великих командах навіть незначні відхилення у стандартах коду або структурі проєкту ускладнюють підтримку системи. Накопичення технічного боргу призводить до уповільнення розробки, підвищення ризику помилок і дублювання функціональності, оскільки значна частина часу витрачається на аналіз застарілих рішень [24]. Додаткові труднощі виникають при масштабуванні SPA, де тісна зв'язаність компонентів ускладнює розподіл відповідальності між командами. Збільшення кодової бази впливає на продуктивність збірки та завантаження, а спільний стан застосунку створює ризик непрогнозованих змін у різних модулях. Навіть локальні модифікації логіки можуть впливати на інші частини системи [24].

Мікрофронтенд-підхід частково зменшує ці проблеми, однак потребує стандартизації взаємодії, стилізації, маршрутизації та обміну даними. Без цього виникає фрагментація системи та складність інтеграції модулів.

Інтеграція AI-компонентів додає нові обмеження, зокрема затримки обробки запитів, залежність від зовнішніх API та обмеження швидкості сервісів. Відсутність кешування та механізмів повторних викликів призводить до нестабільності взаємодії. Додаткове навантаження створюється при виконанні обчислень на клієнтській стороні [25].

Різниця у циклах розробки між фронтендом і AI/ML-компонентами спричиняє асинхронність інтеграції, що призводить до нестабільних сценаріїв і зниження якості продукту. Відсутність єдиних стандартів і управління залежностями перетворює розробку на фрагментований процес із підвищеним рівнем дублювання логіки та складністю внесення змін [26].

У сукупності ці фактори вказують на обмежену ефективність класичних архітектур у сучасних умовах. Зростання масштабу систем і поява AI-інтеграцій вимагають адаптивних архітектурних підходів, здатних підтримувати багаторівневу структуру команд, гнучку інтеграцію компонентів і стабільність системи при швидких змінах.

1.4 Визначення системи метрик для оцінки архітектурних рішень

1.4.1 Значення системи метрик та їх роль у фронтенд-розробці

Оцінка ефективності архітектурних рішень у фронтенді базується на системі об’єктивних метрик, що формалізують аналіз, забезпечують прозорість рішень і дозволяють порівнювати різні підходи. Вони дають можливість виявляти вузькі місця, оцінювати вплив змін і прогнозувати поведінку системи при зростанні навантаження або масштабуванні функціональності.

Метрики є ключовими при виборі між SPA, SSR і SSG, оскільки дозволяють об’єктивно оцінити продуктивність, час до інтерактивності та чутливість інтерфейсу. Вони також використовуються для аналізу ефективності оптимізацій, змін у кодовій базі та оновлень технологічного стеку, забезпечуючи прийняття рішень на основі кількісних показників, а не суб’єктивних оцінок [27].

Система метрик визначає пріоритети оптимізації, підтримує керованість архітектури та знижує ризики накопичення технічного боргу. Її застосування є критичним для великих і довготривалих проєктів, де важлива стабільність, передбачуваність і контроль якості [28].

Крім цього, метрики забезпечують безперервний моніторинг стану системи та дозволяють виявляти потенційні проблеми на ранніх етапах. Це дає змогу своєчасно приймати архітектурні рішення, мінімізувати ризики та підтримувати ефективність системи при зміні навантаження, функціональності або структури команди.

1.4.2 Метрики продуктивності та розробницького досвіду

Оцінка продуктивності фронтенд-застосунку є одним із ключових аспектів аналізу архітектурних рішень. Метрики продуктивності дозволяють кількісно визначити швидкість відгуку інтерфейсу, плавність анімацій, стабільність відображення контенту та здатність системи обробляти дії користувача без затримок. До основних показників належать: Largest Contentful Paint (LCP) — час від завантаження сторінки до
відображення основного візуального контенту. Високий LCP може свідчити про повільне завантаження важливих ресурсів або неоптимальну організацію рендерингу. [29] First Input Delay (FID) — затримка між першою взаємодією користувача (клік, прокрутка, введення тексту) та реакцією інтерфейсу. FID демонструє чутливість системи до дій користувача. Cumulative Layout Shift (CLS) — показник стабільності розташування елементів на сторінці. Великий CLS негативно впливає на UX, оскільки елементи зміщуються під час завантаження. Time to Interactive (TTI) — час, коли сторінка стає повністю інтерактивною, тобто користувач може без затримок виконувати дії. Frames per Second (FPS) — частота оновлення кадрів під час анімацій та прокрутки, що визначає плавність відображення та комфорт користувача. API latency — затримка у відповіді зовнішніх сервісів або бекенду, що безпосередньо впливає на швидкість завантаження динамічного контенту. [30]

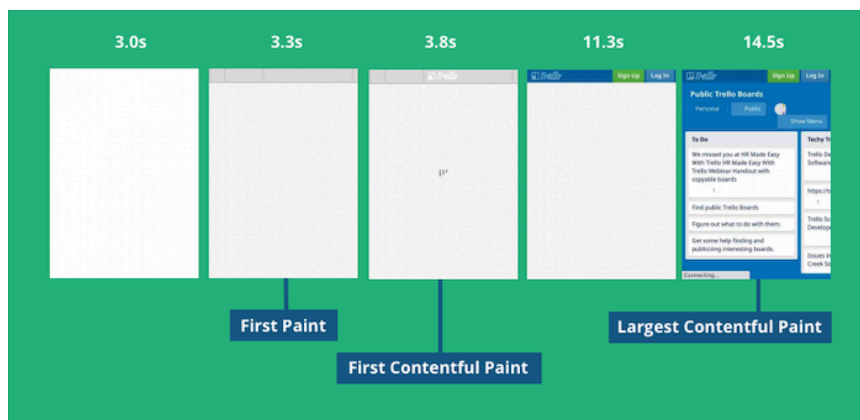


Рис. 1.4.1 - Приклад як вираховується LCP

Швидкодія фронтенд-застосунку безпосередньо впливає на користувацький досвід: повільні або нестабільні інтерфейси призводять до фрустрації користувачів, збільшення показників відмов та зниження конверсії. Відповідно, архітектура повинна враховувати оптимізацію потоків даних, використання кешування, lazy loading, баланс між клієнтським та серверним рендерингом, а також розподілення ресурсів для забезпечення високої продуктивності. [31]

Не менш важливою є оцінка Developer Experience (DX) — зручності та ефективності роботи команди розробників. Метрики DX дозволяють оцінити, наскільки швидко та безболісно новий розробник може включитися у проєкт та працювати з архітектурою. Основні показники включають:

time-to-setup — час, необхідний для повного налаштування локального середовища та запуску застосунку; build time — час збірки застосунку під час розробки або перед розгортанням, що впливає на швидкість ітерацій та тестування; CI/CD latency — затримка виконання процесів автоматичної інтеграції, тестування та деплою; quality of documentation — повнота та зрозумілість внутрішньої документації, що забезпечує легкість адаптації нових розробників та зменшує ймовірність помилок при підтримці системи.

Поєднання метрик продуктивності та DX дозволяє архітекторам і менеджерам приймати обґрунтовані рішення щодо оптимізації архітектури, інструментів розробки та процесів CI/CD, що забезпечує ефективність як кінцевого продукту, так і процесу його створення. [32]

1.4.3 Метрики підтримуваності та масштабованості

Оцінка підтримуваності визначає, наскільки легко фронтенд-застосунок можна розширювати, змінювати та тестувати. Вона базується на метриках, що відображають складність коду, стабільність системи та ризики технічного боргу, зокрема code churn, test coverage і cyclomatic complexity. Високий рівень code churn може свідчити про нестабільність архітектури, тоді як

достатнє покриття тестами знижує ризик помилок і спрощує рефакторинг, а зростання складності ускладнює підтримку коду [33].

Рівень підтримуваності безпосередньо впливає на ефективність розробки: низькі показники призводять до уповільнення роботи, зростання кількості помилок і накопичення технічного боргу, тоді як високі забезпечують швидке внесення змін і стабільність системи [34].

Оцінка масштабованості визначає здатність системи ефективно працювати при зростанні навантаження, функціоналу та кількості команд і включає такі характеристики, як обробка навантаження, модульність і незалежність розгортання. Незалежне розгортання є особливо важливим для мікрофронтенд-архітектур, де окремі частини системи розвиваються автономно [35].

Модульність та ізоляція компонентів забезпечують ефективне масштабування, дозволяючи розділяти систему на незалежні частини для паралельної розробки та розгортання без конфліктів, що підтримує стабільну продуктивність при зростанні системи [36].

Поєднання метрик підтримуваності та масштабованості забезпечує обґрунтоване прийняття архітектурних рішень щодо структури коду, організації модулів і планування розробки, підвищуючи стійкість і ефективність фронтенд-системи.

1.4.4 Використання системи метрик для оптимізації архітектури

Система метрик є ключовим інструментом оцінки архітектурних рішень у фронтенді, оскільки забезпечує перехід від суб’єктивних суджень до кількісного аналізу. Вона дозволяє визначати ефективність окремих аспектів архітектури та виявляти зони, що потребують оптимізації. Метрики використовуються для вдосконалення існуючих рішень: показники продуктивності допомагають знаходити вузькі місця в рендерингу та взаємодії, метрики підтримуваності — виявляти складність коду і проблеми з тестуванням, а метрики масштабованості — оцінювати готовність системи до

зростання та незалежного розгортання модулів [37]. Вони також застосовуються для адаптації архітектури до нових вимог. Показники модульності та deployment independence дозволяють оцінити складність інтеграції нового функціоналу, а метрики Developer Experience (DX) — оптимізувати CI/CD-процеси та прискорити адаптацію розробників.

Для системного підходу до дослідження та оптимізації архітектури корисно класифікувати метрики за основними групами:

- Продуктивніст (Performance): LCP, FID, CLS, TTI, FPS, API latency.
- Підтримуваність (Maintainability): code churn, test coverage, cyclomatic complexity.
- Масштабованість (Scalability): load handling, modularity, deployment independence.

Досвід розробника (Developer Experience, DX): time-to-setup, build time, CI/CD latency, quality of documentation.

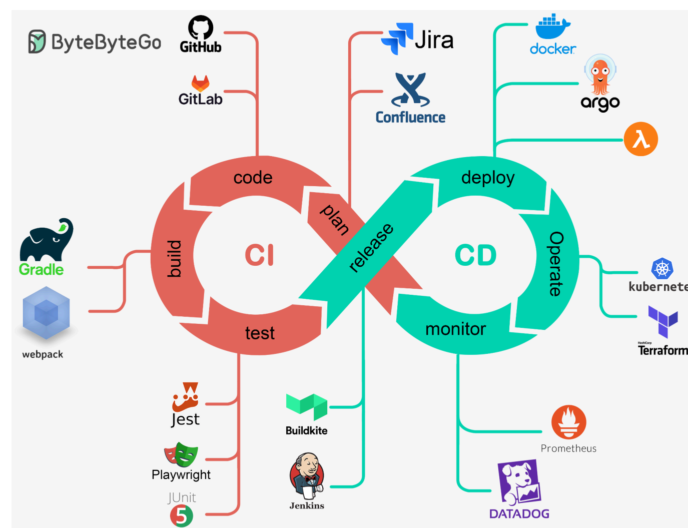


Рис. 1.4.2 - приклад взаємодії CI/CD

Інтеграція цих метрик у регулярний процес оцінки архітектури дозволяє не лише відстежувати поточний стан системи, а й прогнозувати її поведінку при зміні навантаження, розширенні функціоналу або збільшенні команди. Використання метрик як інструменту оптимізації сприяє створенню

стабільних, гнучких та довготривалих архітектурних рішень, які відповідають як технічним, так і бізнес-вимогам.

1.5 Формування методичних засад подальшого дослідження і постановка завдань для розробки власного підходу

Після детального аналізу сучасних тенденцій фронтенд-розробки, критеріїв вибору архітектури та проблем, що виникають у великих командах і при інтеграції AI-компонентів, постає необхідність переходу від опису існуючих підходів до формування власного наукового дослідження. Логіка цього переходу полягає у систематизації отриманих знань: на основі аналізу переваг і недоліків різних архітектурних моделей можна визначити, які аспекти варто зберегти, а які потребують адаптації чи поліпшення. Методологічна база дозволяє не лише формувати гіпотези щодо ефективності архітектури, а й обґрунтовано планувати експериментальні дослідження. [38]

Для проведення такого дослідження планується використати комплексний підхід, що поєднує порівняльний аналіз, моделювання та тестування на практичних прикладах. Порівняльний аналіз включатиме вивчення SPA, SSR, SSG, Islands Architecture, мікрофронтендів і серверних компонентів з точки зору продуктивності, підтримуваності, масштабованості та досвіду розробника. Методи моделювання дозволять створити узагальнену модель оцінки архітектур, де ключові показники з попереднього підпункту (Performance, Developer Experience, Maintainability, Scalability) будуть інтегровані у систему для об’єктивного порівняння. Практичне тестування на кейсах допоможе перевірити придатність моделі в реальних умовах, визначити слабкі місця і скорегувати архітектурні рішення до оптимального стану.

Завдання дослідження можна сформулювати так: спочатку необхідно розробити модель оцінки архітектурних рішень, яка враховує показники продуктивності, зручності роботи команди, підтримуваності та

масштабованості. Далі модель перевіряється на конкретних кейсах, де відображаються сценарії з великими SPA, інтеграцією AI-компонентів і роботою кількох команд. Наступним етапом є порівняння отриманих результатів з існуючими підходами та виявлення сильних і слабких сторін, що дозволяє коригувати рекомендації щодо побудови адаптивної архітектури. [39]

Очікувані результати дослідження включають розробку системи критеріїв і метрик, які полегшують вибір архітектурного підходу для різних типів фронтенд-застосунків, зокрема тих, що використовують AI-компоненти. Завдяки аналітичному підходу можна буде визначити, які рішення забезпечують оптимальний баланс між продуктивністю, гнучкістю та зручністю для розробників, а які варто модифікувати. Результати також слугуватимуть основою для подальшої розробки методології ARX, де буде поєднано сильні сторони різних архітектур із врахуванням специфіки сучасних командних процесів та потреб AI-застосунків.

Важливо зазначити, що методичні засади дослідження орієнтовані на практичне застосування. Вони передбачають, що кожен етап — від моделювання до тестування на кейсах — інтегрується у систему контролю якості та аналітики, що дозволяє відстежувати зміни продуктивності, підтримуваності та досвіду розробника у реальному часі. Такий підхід створює основу для формування власного архітектурного підходу, який зможе адаптуватися до різних типів проектів, кількості команд та інтеграції AI-компонентів, а також забезпечувати прогнозовану стабільність і масштабованість системи. [40]

На підставі викладеного формується напрям власного підходу до архітектури фронтенд-застосунків, який поєднує аналітичні методи, моделювання, практичне тестування і адаптивність під потреби команд та сучасних технологій. Цей підхід стане логічним переходом до наступного розділу роботи, де планується порівняльний аналіз існуючих архітектурних моделей та обґрунтування створення методології ARX.

1.6 Висновки до першого розділу

У розділі виконано аналіз сучасних вимог до архітектур фронтенд-застосунків та визначено ключові підходи, що використовуються на практиці, зокрема SPA, SSR, SSG і мікрофронтенди. Встановлено, що жоден із підходів не забезпечує універсального рішення: SPA забезпечує високу інтерактивність, але має проблеми з продуктивністю та SEO; SSR і SSG покращують початкове завантаження, однак ускладнюють оновлення даних; мікрофронтенди підвищують масштабованість, але збільшують складність координації. Також підтверджено, що ефективність архітектури безпосередньо залежить від збалансованого врахування продуктивності, підтримуваності, масштабованості та Developer Experience.

Отож визначення системи метрик дозволяє об’єктивно оцінювати рішення, виявляти вузькі місця та обґрунтовувати архітектурні зміни. Отримані результати формують основу для розробки адаптивної архітектури, здатної комбінувати різні підходи залежно від контексту використання, що є критично важливим для сучасних масштабованих і динамічних вебсистем.

2. ПОРІВНЯЛЬНИЙ АНАЛІЗ ІСНУЮЧИХ АРХІТЕКТУРНИХ МОДЕЛЕЙ ФРОНТЕНДУ ТА ПЕРЕДУМОВИ СТВОРЕННЯ МЕТОДОЛОГІЇ ARX

2.1. Модель SPA

Архітектурна модель Single Page Application (SPA) ґрунтується на ідеї побудови вебзастосунку, що працює в межах однієї HTML-сторінки, де контент оновлюється динамічно без повного перезавантаження сторінки. Це принципово відрізняє SPA від класичних багатосторінкових застосунків (Multi Page Applications, MPA), у яких кожен запит користувача до нової сторінки ініціює повну відповідь сервера з новим HTML-документом. У SPA браузер завантажує базовий каркас сайту один раз, після чого більшість дій виконується на клієнтській стороні, що забезпечує плавну та швидку взаємодію.

2.1.1 Принцип роботи SPA

Основним елементом, який забезпечує таку поведінку, є client-side routing — механізм маршрутизації, що реалізується за допомогою JavaScript. Замість традиційного переходу між окремими HTML-сторінками, маршрутизатор (наприклад, React Router або Vue Router) відстежує зміни в URL і оновлює лише ту частину інтерфейсу, яка потребує перерендерингу. Це дозволяє уникнути зайвих запитів до сервера, оскільки зміни відбуваються всередині браузера, а не через повторне завантаження документа. Таким чином, користувач отримує миттєвий відгук системи, що значно підвищує зручність використання. [41]

Ключову роль у цьому процесі відіграють сучасні вебтехнології, зокрема AJAX, Fetch API, Virtual DOM та WebSocket. AJAX і Fetch API забезпечують асинхронну передачу даних між клієнтом і сервером, дозволяючи оновлювати частини сторінки без її повного перезавантаження.

Virtual DOM (віртуальна модель об’єктів документа), реалізована у фреймворках React та Vue.js, мінімізує кількість реальних оновлень у DOM, що суттєво підвищує продуктивність інтерфейсу. WebSocket, своєю чергою, забезпечує двосторонню комунікацію в реальному часі, що дає можливість SPA-архітектурам підтримувати постійне оновлення даних — наприклад, у чатах, панелях моніторингу або аналітичних системах. [42]

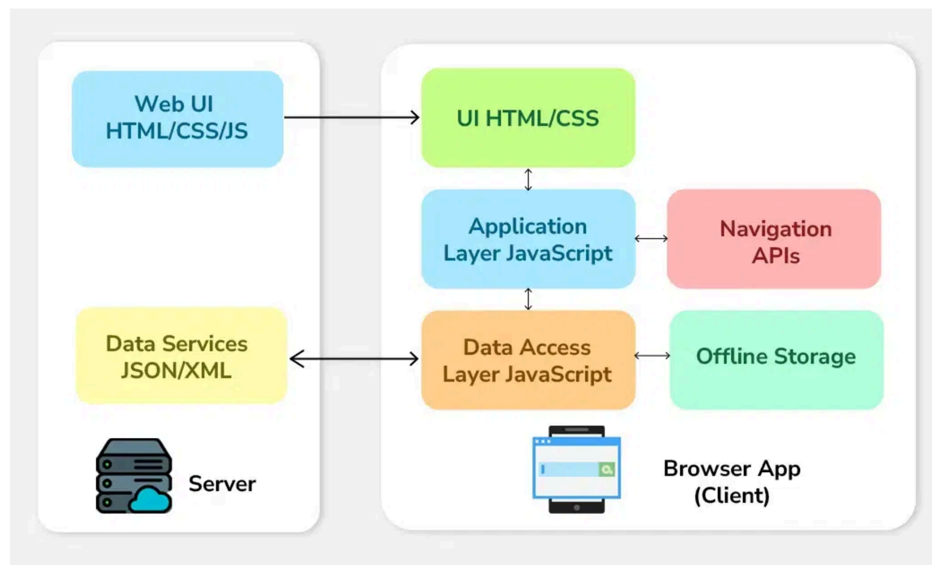


Рис. 2.1.1 - Модель SPA

SPA також передбачає чітке розділення фронтенду та бекенду. Клієнтська частина (front-end) відповідає за рендеринг інтерфейсу, керування станом і логіку взаємодії користувача, тоді як серверна (back-end) виступає лише як джерело даних. Комунікація між цими частинами відбувається через REST API або GraphQL API, що забезпечує незалежність між ними. Такий підхід сприяє масштабуванню, адже один і той самий бекенд може обслуговувати різні клієнтські застосунки — веб, мобільні або десктопні.

Популярні фреймворки — React, Angular та Vue.js — реалізують SPA-модель за допомогою компонентного підходу. У React структура застосунку будується на функціональних або класових компонентах, які відповідають за певну частину інтерфейсу та можуть перевикористовуватися у різних частинах програми. Angular використовує модульну архітектуру з

шаблонною системою та двостороннім зв’язком даних (two-way data binding), що спрощує інтерактивні оновлення. Vue.js, своєю чергою, забезпечує просту декларативну модель і реактивність через систему спостереження за змінами стану. [43]

Важливою складовою роботи SPA є управління станом (state management). Коли застосунок містить сотні компонентів, потрібно гарантувати синхронізацію даних між ними. Для цього використовуються спеціальні бібліотеки, такі як Redux у React, NgRx в Angular або Pinia/Vuex у Vue.js. Вони дозволяють централізовано зберігати стан застосунку та уникати конфліктів між різними частинами інтерфейсу.

Архітектура SPA створює відчуття роботи з нативним застосунком, оскільки інтерфейс реагує миттєво, а зміни відбуваються без видимих перезавантажень сторінки. Завдяки mobile-first підходу та ефективній маршрутизації, SPA забезпечує плавний користувацький досвід (UX), що нагадує взаємодію з мобільними або десктопними програмами. Саме ця характеристика зробила SPA домінуючою моделлю у веброзробці останнього десятиліття, оскільки вона поєднує швидкість, інтерактивність та модульність у єдиній архітектурі. [44]

2.1.2 Продуктивність і UX у SPA

Однією з ключових переваг архітектури Single Page Application (SPA) є висока продуктивність і швидка реакція інтерфейсу після початкового завантаження. Завдяки тому, що SPA працює в межах однієї сторінки, усі подальші дії користувача виконуються локально на клієнті, без необхідності повторних повних запитів до сервера. Це дає змогу забезпечити миттєву взаємодію та створює враження безперервної роботи застосунку. Такий підхід є особливо ефективним для інтерфейсів, що вимагають швидкої зміни станів — чатів, поштових клієнтів, панелей керування чи CRM-систем.

Продуктивність SPA значною мірою зумовлена скороченням кількості серверних запитів. На відміну від традиційних багатосторінкових застосунків

(MPA), де кожен перехід між сторінками передбачає повне завантаження нового HTML-документа, SPA після першого запуску отримує лише необхідні дані через асинхронні API-запити. Це дозволяє передавати значно менше інформації між клієнтом і сервером, оскільки структура сторінки вже завантажена, а оновлюється лише контент. Як результат, SPA істотно зменшує затримку відповіді системи, що особливо важливо для сучасних користувачів, які очікують максимально швидкої реакції. [45]

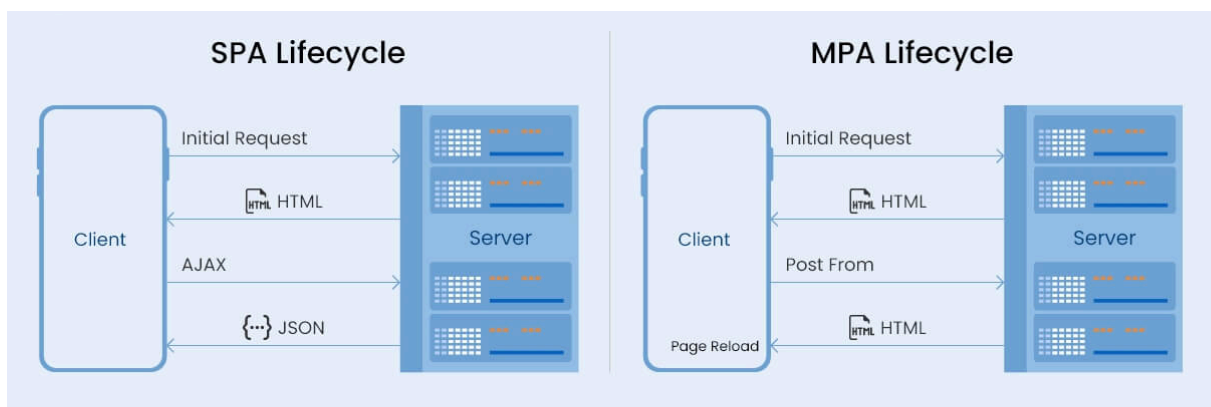


Рис. 2.1.2 - Порівняльна характеристика SPA та MPA

Щоб підтримувати високу швидкодію, SPA використовує комплекс технік оптимізації. Однією з них є кешування — збереження статичних ресурсів (зображень, шрифтів, стилів, скриптів) у браузері для їхнього повторного використання без повторного завантаження. Сучасні застосунки застосовують Service Workers, які забезпечують контроль над кешем і дозволяють працювати навіть у офлайн-режимі, коли мережеве з'єднання недоступне. Іншим поширеним підходом є prefetching — попереднє завантаження ресурсів, які можуть знадобитися користувачу на наступних етапах роботи. Це зменшує затримку при переходах між розділами.

Важливу роль відіграє lazy loading, що передбачає завантаження лише тих компонентів і модулів, які потрібні користувачу в даний момент. Це значно скорочує час першого рендерингу. Технологія code-splitting, реалізована, наприклад, у Webpack або Vite, ділить JavaScript-код на менші частини

(chunks), які довантажуються за потреби. Таким чином, SPA може підтримувати складну логіку та велику кількість компонентів, не жертвуючи продуктивністю. [46]

Завдяки таким оптимізаціям SPA забезпечує високий рівень користувацького досвіду (UX). Інтерфейс працює плавно, без затримок або миготінь сторінки, що створює враження нативної програми. Відсутність постійних перезавантажень дозволяє реалізувати безперервну навігацію — користувач переходить між розділами майже миттєво. Крім того, SPA підтримує реактивність — інтерфейс динамічно реагує на зміну стану даних, що робить взаємодію інтуїтивною та передбачуваною. Завдяки кешуванню та офлайн-підтримці користувач може продовжувати роботу навіть без активного інтернет-з’єднання, а дані синхронізуються автоматично після відновлення мережі.

Проте, незважаючи на численні переваги, SPA має й певні проблеми, пов’язані з початковим завантаженням. Головним викликом є великий розмір початкового JavaScript-бандлу, який містить код усього застосунку. Це призводить до збільшення часу Time to First Paint (TTFP) та Largest Contentful Paint (LCP) — метрик, які безпосередньо впливають на сприйняття швидкості завантаження користувачем. Особливо помітним це стає на мобільних пристроях із повільним з’єднанням. Щоб пом’якшити цю проблему, розробники впроваджують оптимізацію рендерингу, зокрема відкладене завантаження скриптів, стиснення ресурсів і динамічне підвантаження сторінок через Suspense у React або Dynamic Imports у Vue.js.

Компроміс між продуктивністю та функціональністю є однією з основних дилем SPA-архітектури. З одного боку, розробники прагнуть досягти максимальної інтерактивності та гнучкості, додаючи численні модулі, бібліотеки й анімації. З іншого — кожен додатковий пакет збільшує розмір бандлу й погіршує швидкість першого завантаження. Оптимальне рішення полягає в балансі між динамікою та ефективністю, який досягається

завдяки розподіленню коду, серверному кешуванню та використанню CDN. [47]

Багато провідних вебзастосунків демонструють ефективність SPA навіть на великих масштабах. Наприклад, Gmail використовує SPA-архітектуру для забезпечення миттєвої взаємодії між поштовими елементами, при цьому застосовуючи агресивне кешування та фонове завантаження листів. Trello реалізує асинхронні оновлення через WebSocket, що забезпечує роботу в реальному часі без перезавантаження сторінки. Slack поєднує SPA з прогресивними механізмами кешування, дозволяючи працювати навіть при тимчасовій втраті мережі.

Таким чином, SPA залишається домінуючою архітектурною моделлю для інтерактивних вебплатформ, завдяки здатності забезпечувати плавний користувацький досвід, швидку реакцію інтерфейсу та багатий функціонал. Попри виклики, пов’язані з початковим завантаженням і складністю оптимізації, ефективні інструменти управління ресурсами, розвинені екосистеми фреймворків і сучасні практики розробки роблять SPA оптимальним вибором для створення масштабних, динамічних і зручних користувацьких застосунків.

2.1.3 Обмеження SEO і шляхи їх вирішення

Одним із головних викликів архітектури Single Page Application (SPA) є її обмежена сумісність із системами пошукової оптимізації (SEO). На відміну від традиційних багатосторінкових сайтів, де сервер надсилає браузеру готовий HTML-документ, SPA генерує більшу частину контенту динамічно на клієнтській стороні за допомогою JavaScript. Це означає, що в початковій версії HTML-файлу, який отримує пошуковий бот, часто відсутній основний контент, мета-теги, заголовки або структуровані дані. Внаслідок цього пошукові системи не можуть коректно індексувати сторінки SPA, що призводить до зниження видимості таких вебзастосунків у результатах пошуку. [48]

Пошукові системи, зокрема Google, використовують вебкраулери, які сканують HTML-код сторінки та визначають її зміст на основі статичної розмітки. У класичних серверних застосунках (наприклад, SSR або SSG) бот одразу отримує повністю сформований HTML-документ з усіма елементами контенту, тому індексація відбувається швидко та ефективно. У випадку ж SPA краулер стикається з тим, що сторінка виглядає «порожньою» до моменту виконання JavaScript-коду. Хоча сучасні пошукові системи здатні рендерити JavaScript, цей процес є ресурсомістким і часто відбувається із затримкою, що призводить до неповної або некоректної індексації.

Рис. 2.1.1 - Схематична діаграма складових SEO

Типовими проблемами SEO у SPA є відсутність мета-тегів у вихідному HTML, динамічне завантаження контенту, а також неможливість відстеження окремих станів сторінок. Наприклад, сторінки, що генеруються динамічно через client-side routing, не мають унікальних HTML-файлів — усі посилання ведуть на одну й ту ж базову сторінку. Це ускладнює побудову індексу пошуковими ботами та негативно впливає на ранжування.

Для подолання цих обмежень у сучасній розробці використовуються кілька стратегій. Одним із найбільш поширених рішень є Server-Side Rendering (SSR), що дозволяє попередньо рендерити сторінку на сервері перед її відправленням користувачу. SSR реалізується у таких фреймворках,

як Next.js для React або Nuxt.js для Vue.js, поєднуючи динамічність SPA з перевагами серверного рендерингу.

Іншим ефективним підходом є Pre-rendering, який передбачає створення статичних HTML-версій сторінок на етапі збірки. Такий метод активно використовується у фреймворках Gatsby або Scully, де сторінки генеруються наперед і зберігаються у вигляді статичних файлів.

Ще одним способом вирішення SEO-проблем у SPA є Dynamic Rendering — технологія, що дозволяє подавати різні версії сторінок користувачам і пошуковим ботам. У цьому випадку сервер визначає тип запиту і, для краулерів, надсилає вже відрендерену версію сторінки.

На практиці багато сучасних фреймворків уже інтегрують ці методи безпосередньо у свої екосистеми. Наприклад, у Next.js реалізовано комбінацію SSR і Static Site Generation , а у Gatsby — повну статичну генерацію. Водночас кожне з цих рішень має свої компроміси: SSR підвищує навантаження на сервер, pre-rendering не підходить для часто змінюваного контенту, а dynamic rendering ускладнює підтримку. Проблеми SEO у SPA стали стимулом до розвитку гібридних архітектур, які поєднують різні підходи до рендерингу.

2.2 Методика порівняння архітектур і вибір метрик

Порівняння різних архітектурних моделей фронтенд-застосунків вимагає системного підходу, який базується не лише на суб’єктивних спостереженнях, а й на об’єктивних кількісних та якісних показниках. Для цього у даному дослідженні використовується методика порівняльного аналізу архітектур на основі чотирьох ключових груп метрик, визначених у розділі 1.4: Performance (продуктивність), Developer Experience (досвід розробника), Maintainability (підтримуваність) та Scalability (масштабованість). Кожна з цих категорій відображає певний аспект

ефективності архітектури, що дозволяє сформувати комплексну оцінку її придатності для різних типів проєктів. [52]

Основним завданням цієї методики є створення порівняльної бази, у межах якої різні архітектурні моделі — SPA, SSR, SSG, Islands, Micro-Frontends та Server Components — аналізуються за єдиними критеріями. Такий підхід дає змогу визначити, яка з них забезпечує найкращий баланс між швидкодією, стабільністю, масштабованістю та зручністю розробки.

Першою категорією є Performance, яка характеризує швидкодію користувацького інтерфейсу та реактивність системи. Вона вимірюється за допомогою кількісних метрик, зокрема Largest Contentful Paint (LCP), Time to Interactive (TTI), First Input Delay (FID), Total Blocking Time (TBT) і Frames per Second (FPS). Ці показники дозволяють оцінити, наскільки швидко користувач бачить контент і може взаємодіяти з ним. Високі значення продуктивності особливо важливі для проєктів, де критичною є швидкість відгуку — наприклад, у торгових платформах, реальних чатах або інтерактивних AI-додатках.

Друга категорія — Developer Experience (DX) — оцінює ефективність роботи команди розробників. Вона включає як технічні, так і організаційні показники: time-to-setup (час налаштування середовища), build time (тривалість збірки застосунку), CI/CD latency (затримка при автоматичному деплої), доступність документації та якість інструментів відлагодження. У сучасних архітектурах, таких як Micro-Frontends або Server Components, ці показники стають вирішальними, оскільки впливають на швидкість розробки, командну синхронізацію та легкість інтеграції нових учасників у проєкт.

Наступна група метрик — Maintainability (підтримуваність) — показує, наскільки просто підтримувати, оновлювати й розширювати кодову базу без ризику дестабілізувати систему. Тут застосовуються показники code churn (частота змін у коді), test coverage (покриття тестами), cyclomatic complexity (цикломатична складність) і якість документації. Висока підтримуваність

особливо важлива для довготривалих корпоративних або урядових проєктів, де архітектура має витримувати багаторазові оновлення, зміни вимог та масштабування без суттєвих втрат якості. [53]

Четверта категорія — Scalability (масштабованість) — характеризує здатність архітектури ефективно працювати при зростанні навантаження, кількості користувачів або функціональних модулів. Основними показниками є load handling (здатність обробляти навантаження), modularity (ступінь розділення системи на незалежні модулі) та deployment independence (можливість окремого розгортання частин системи). Ці метрики особливо важливі для великих продуктів із розподіленою розробкою, де одночасно працюють десятки команд.

Обґрунтування вибору саме цих критеріїв полягає в тому, що вони охоплюють усі основні аспекти архітектурної ефективності — користувацький досвід, внутрішню якість коду, організаційну гнучкість і технічну стабільність. У сукупності ці показники створюють повну картину не лише з точки зору технології, але й із позиції управління розробкою, що робить їх універсальними для оцінки як стартапів, так і масштабних корпоративних платформ.

Використання цієї методики дозволяє також класифікувати архітектури за типами застосувань. Наприклад, SPA найкраще підходять для високодинамічних інтерфейсів і мобільних застосунків, де головну роль відіграє UX. SSR і SSG оптимальні для SEO-орієнтованих платформ, маркетплейсів і контентних сайтів, де важлива початкова швидкодія. Micro-Frontends і Server Components забезпечують масштабованість та ізоляцію команд у великих проєктах, а Islands Architecture поєднує ці підходи, створюючи адаптивні системи, що динамічно розподіляють навантаження. [54]

У результаті така методика порівняння дає можливість не лише виявити сильні та слабкі сторони кожної архітектури, а й сформувати підґрунтя для подальшого проектування нової адаптивної архітектурної моделі, яка

об'єднає найкращі властивості існуючих підходів. Саме на цій основі в наступному підрозділі буде побудовано порівняльну таблицю архітектур фронтенду.

2.2.1. Порівняльна таблиця архітектур

Сучасні архітектурні моделі фронтенду, такі як SPA, SSR, SSG, Islands, Micro-Frontends та Server Components, відрізняються за продуктивністю, SEO, підтримуваністю, масштабованістю та Developer Experience (DX). Кожна з цих моделей формує певний баланс між швидкістю взаємодії користувача, гнучкістю розробки та простотою масштабування, і їх вибір залежить від специфіки проєкту, команди та бізнес-вимог.

SPA (Single Page Application) характеризується високою інтерактивністю і миттєвою реакцією інтерфейсу після першого завантаження. Вона дозволяє зменшити кількість запитів до сервера завдяки client-side routing та динамічному оновленню контенту. Основні переваги SPA — безперервна навігація, плавна взаємодія та компонентна структура інтерфейсу. Однак велика кількість JavaScript-коду на старті створює повільне початкове завантаження і ускладнює SEO-оптимізацію. Приклади великих продуктів на SPA включають Gmail, Trello та Slack, де застосовані методи оптимізації, як lazy loading та code-splitting, для забезпечення стабільної продуктивності. [55]

SSR (Server-Side Rendering) передбачає попереднє формування HTML на сервері, що значно скорочує час до першого рендерингу (LCP) і покращує індексацію пошуковими системами. SSR дозволяє забезпечити швидкий доступ до контенту навіть на слабких пристроях або при повільному інтернет-з'єднанні. Складність підтримки та налаштування CI/CD у SSR вища, ніж у SPA, але такий підхід підходить для контентно-орієнтованих платформ, таких як Next.js. Основна перевага SSR — збалансованість між продуктивністю та SEO, а недоліком є підвищене навантаження на сервер при великій кількості користувачів.

SSG (Static Site Generation) генерує статичні HTML-сторінки під час збірки, що дозволяє досягти максимальної швидкості рендерингу та високої SEO-дружності. SSG добре підходить для невеликих та середніх проєктів або документаційних сайтів, де контент змінюється рідко. Оновлення даних у великих проєктах є більш складним через необхідність повторної генерації всіх сторінок. Приклади використання SSG — Gatsby та Hugo, де застосовується ефективна кешова політика та CDN для швидкого розподілу контенту. Основний недолік — менша гнучкість при динамічному контенті порівняно з SPA та SSR. [56]

Islands Architecture поєднує переваги статичного рендерингу з інтерактивними компонентами (“островами”), що дозволяє прискорити TTI та покращити SEO без втрати динамічності. Кожен інтерактивний компонент завантажується окремо, що зменшує обсяг JavaScript і підвищує продуктивність. Така архітектура добре підходить для великих магазинів і платформ з активним користувацьким інтерфейсом, приклад — Shopify Hydrogen. Основні переваги — модульність та швидке оновлення інтерактивних компонентів, а недолік — збільшена складність побудови та тестування окремих “островів”.

Micro-Frontends дозволяють розділити фронтенд на незалежні модулі, які можуть розроблятися різними командами паралельно. Це підвищує масштабованість, спрощує підтримку великих проєктів і дозволяє незалежне розгортання модулів. Основні переваги — гнучка архітектура, ізоляція компонентів та спрощене масштабування, недоліки — складніший DX та інтеграція модулів. Приклади великих систем — Spotify та Amazon, де застосування Micro-Frontends дозволяє командам працювати автономно над різними частинами продукту.

Server Components виконують рендеринг частин інтерфейсу на сервері, що зменшує розмір JavaScript-бандлів і прискорює перший рендеринг. Це сучасний підхід для гібридних рішень, поєднуючи динамічність SPA з продуктивністю SSR. Server Components дозволяють оптимізувати

продуктивність, SEO та масштабованість одночасно, але вимагають складної інфраструктури та підтримки серверного рендерингу. Приклади реалізації — нові проєкти на Next.js 13+ з використанням React Server Components.

2.3 Архітектура ARX

Adaptive Rendering eXperience (ARX) є сучасною адаптивною архітектурою фронтенду, яка інтегрує ключові переваги існуючих підходів: SPA, SSR, SSG, Islands Architecture, Micro-Frontends та Server Components. Головною метою ARX є забезпечення оптимального балансу між продуктивністю, SEO, підтримуваністю, масштабованістю та Developer Experience (DX) у проєктах із високим рівнем інтерактивності та динамічності. Архітектура дозволяє поєднувати швидку реакцію інтерфейсу, ефективну роботу з динамічним контентом та можливість масштабування великих систем.

2.3.1. Концепція та цілі ARX

Ключовою особливістю ARX є адаптивний підхід до рендерингу компонентів та сторінок. Для кожного елемента системи автоматично підбирається оптимальний режим рендерингу залежно від навантаження, характеру контенту та пристрою користувача. Наприклад, статичні сторінки можуть генеруватися за принципом SSG для забезпечення максимальної швидкості завантаження та SEO-дружності, інтерактивні панелі — через Islands Architecture або Server Components для мінімізації JavaScript-бандлів, а динамічні сторінки з високим рівнем взаємодії — як SPA з client-side routing. Крім того, модульний підхід Micro-Frontends дозволяє паралельну розробку окремих компонентів різними командами та їхнє незалежне розгортання, що підвищує масштабованість системи.

ARX особливо ефективна у проєктах з ІА-згенерованими фронтенд-застосунками, де структура інтерфейсу та компоненти можуть динамічно змінюватися. Використання ARX дозволяє автоматично

адаптувати тип рендерингу для ІА-компонентів залежно від їх функціональної ролі, обсягу даних та частоти оновлення. Це забезпечує високу продуктивність, стабільність інтерфейсу та комфорт для розробників, які інтегрують автоматично згенеровані компоненти. У результаті ARX створює архітектурне середовище, яке гнучко реагує на зміни вимог, забезпечує безперервну інтерактивність та зменшує технічний борг у процесі розвитку ІА-фронтенд-платформ. [57]

2.3.2. Адаптивний рендеринг у ARX

Адаптивний рендеринг є ключовим механізмом ARX, який забезпечує оптимальний режим відображення контенту для кожної сторінки або окремого компонента. ARX визначає, чи слід застосувати статичний рендеринг (SSG), серверний рендеринг (SSR) або клієнтський рендеринг (SPA) залежно від набору критеріїв, таких як навантаження на сервер, тип контенту, пристрій користувача та важливість SEO для конкретного елемента. Статичний рендеринг використовується для сторінок із рідко змінюваним контентом, що дозволяє досягти максимальної швидкості завантаження та високої SEO-дружності. Серверний рендеринг оптимальний для динамічного контенту, який потребує швидкого доступу користувачів та забезпечення правильного відображення для пошукових ботів. Клієнтський рендеринг застосовується для інтерактивних компонентів з високою частотою оновлення даних, забезпечуючи безперервну навігацію та високий рівень UX.

Крім вибору типу рендерингу, ARX адаптує рендеринг у реальному часі, враховуючи навантаження на сервер та продуктивність клієнтського пристрою. Наприклад, при високій кількості одночасних користувачів система може перевести менш критичні компоненти на статичний рендеринг або Islands Architecture для зменшення обсягу JavaScript-бандлів та пришвидшення Time to Interactive (TTI). Одночасно ARX забезпечує оптимізацію показника Largest Contentful Paint (LCP) для ключових елементів

сторінки, що підвищує сприйняття швидкості користувачем і покращує загальний UX. [58]

Таким чином, адаптивний рендеринг у ARX дозволяє балансувати між продуктивністю, SEO та інтерактивністю, створюючи архітектуру, яка автоматично підбирає оптимальний режим відображення для кожного компонента або сторінки. Такий підхід знижує навантаження на сервер, скорочує час рендерингу та підтримує високу гнучкість фронтенд-системи, особливо у проєктах із ІА-згенерованим контентом.

2.3.3. Застосування ARX у ІА-фронтенді

ARX знаходить особливе застосування у проєктах, де інтерфейси генеруються штучним інтелектом, забезпечуючи адаптивну інтеграцію автоматично створюваних компонентів у фронтенд. Завдяки гнучкому підходу до рендерингу, кожен ІА-компонент може підбирати оптимальний режим відображення — статичний для рідко змінюваних елементів, серверний для динамічного контенту або клієнтський для інтерактивних панелей. Це дозволяє ефективно працювати з великими обсягами даних, персоналізованими дашбордами та інтерфейсами з високою частотою оновлення інформації. [59]

ARX забезпечує високий рівень масштабованості та гнучкості, що критично для ІА-фронтенду, де компоненти часто генеруються та змінюються автоматично. Команди розробників отримують можливість швидкого старту, легко інтегрувати нові ІА-компоненти та працювати паралельно над різними модулями без ризику конфліктів. Крім того, модульна структура ARX дозволяє оновлювати окремі компоненти без повного перезавантаження системи, що зменшує час розробки та покращує стабільність продукту. [60]

Завдяки таким властивостям, ARX створює ефективне середовище для фронтенд-застосунків з ІА-контентом, поєднуючи адаптивність, високу продуктивність та зручність розробки, одночасно підтримуючи високу якість UX та оптимізовану SEO-структуру сторінок.

2.4 Висновки до другого розділу

Проведений аналіз архітектур фронтенду підтверджує, що кожен підхід формує власний баланс між продуктивністю, SEO, підтримуваністю, масштабованістю та Developer Experience. SPA забезпечує високий рівень інтерактивності, але поступається у швидкості початкового завантаження та SEO; SSR і SSG покращують перший рендеринг і доступність для пошукових систем, проте ускладнюють оновлення даних і процеси розгортання; сучасні підходи, такі як Islands Architecture, Micro-Frontends і Server Components, підвищують модульність і масштабованість, але вимагають більш складної координації. У результаті встановлено, що вибір архітектури завжди пов’язаний із компромісами та залежить від конкретних вимог проєкту.

Отримані результати обґрунтовують доцільність використання адаптивної методології ARX, яка дозволяє комбінувати різні моделі рендерингу залежно від контексту використання. Такий підхід забезпечує одночасне покращення продуктивності, користувацького досвіду та SEO, а також підвищує гнучкість розробки за рахунок модульності та можливості паралельної роботи команд. Таким чином, ARX виступає як ефективна архітектурна модель для створення сучасних вебзастосунків, орієнтованих на масштабованість, адаптивність і автоматизацію процесів розробки.

3. ОПТИМІЗАЦІЯ ТА ПРАКТИЧНА РЕАЛІЗАЦІЯ АРХІТЕКТУРНОЇ МЕТОДОЛОГІЇ ARX

3.1. Архітектурна концепція ARX: гібридна модель рендерингу та принципи адаптивності

Архітектурна концепція ARX це узагальненена модель побудови вебзастосунків із динамічними даними, орієнтовану на забезпечення високої продуктивності, гнучкості та масштабованості. Концепція ARX розглядається як інтегрована система керування процесом рендерингу, у межах якої прийняття рішень щодо способу відображення інтерфейсу здійснюється на основі сукупності контекстних параметрів. Такий підхід дозволяє формалізувати процес побудови інтерфейсу як адаптивну архітектурну задачу.

Незважаючи на розвиток сучасних підходів до рендерингу, вони здебільшого розглядаються як ізольовані стратегії, що обираються на рівні фреймворку або інфраструктури, а не як динамічно керовані архітектурні рішення. Це призводить до відсутності єдиного механізму адаптації рендерингу в межах однієї системи. У результаті вибір стратегії часто фіксується на етапі розробки і не змінюється під час виконання, що обмежує ефективність використання ресурсів.

Ключовою складовою ARX є гібридна модель рендерингу, що передбачає одночасне використання серверного (SSR), клієнтського (CSR) та інкрементального або часткового рендерингу (ISR/partial). У межах цієї моделі кожен інтерфейсний компонент розглядається як незалежна одиниця, для якої може бути обрана оптимальна стратегія відображення. Серверний рендеринг забезпечує швидке формування початкового стану інтерфейсу, клієнтський — підтримує інтерактивність і динамічну взаємодію, тоді як інкрементальний підхід дозволяє оновлювати лише окремі частини інтерфейсу без повного перерахунку всієї структури. Таким чином, існує

потреба у формалізації єдиного підходу, який дозволяє розглядати рендеринг як керований процес із можливістю динамічного вибору стратегії залежно від контексту виконання.

Поєднання зазначених підходів реалізується через механізм узгодженого керування рендерингом, який забезпечує розподіл відповідальності між різними рівнями системи. Вибір конкретної стратегії для кожного компонента здійснюється динамічно, з урахуванням характеристик даних, частоти їх оновлення та вимог до швидкодії. Така організація дозволяє мінімізувати затримки відображення та оптимізувати використання обчислювальних ресурсів.

Важливим аспектом концепції ARX є принцип адаптивності, який передбачає здатність архітектури змінювати свою поведінку залежно від умов виконання. Адаптивність реалізується шляхом врахування контексту використання, включаючи параметри клієнтського пристрою, обсяг і тип даних, а також вимоги до продуктивності. У межах цього підходу архітектура забезпечує автоматизоване прийняття рішень щодо доцільності використання певної стратегії рендерингу для конкретного сценарію. Додатково, концепція ARX передбачає використання узагальнених правил і політик, які визначають залежності між характеристиками компонентів і відповідними стратегіями їх обробки. Це дозволяє забезпечити узгодженість поведінки системи та спростити масштабування застосунку за рахунок повторного використання архітектурних рішень.

Наукова новизна запропонованої концепції полягає у формалізації гібридного рендерингу як керованого процесу, що інтегрується безпосередньо в архітектурний рівень системи. На відміну від підходів, де вибір стратегії рендерингу є статичним або фрагментованим, у ARX запропоновано модель контекстно-адаптивного прийняття рішень, яка враховує сукупність параметрів середовища виконання та характеристик компонентів. Це дозволяє розглядати процес рендерингу як динамічну оптимізаційну задачу та створює передумови для подальшої автоматизації

архітектурних рішень.

Отже, запропонована модель ARX формує перехід від статичного вибору архітектурної стратегії рендерингу до контекстно-керованої системи прийняття рішень, що дозволяє розглядати фронтенд-архітектуру як адаптивну систему керування.

3.2. Компоненти системи ARX: рендер-оркестратор, профайлер, build-пайплайн, система політик, AI-генерація scaffold.

У межах архітектурної концепції ARX виділено набір ключових компонентів, які забезпечують реалізацію гібридної моделі рендерингу та принципів адаптивності. Кожен із компонентів виконує окрему функціональну роль, проте їх сукупна взаємодія формує цілісну систему керування життєвим циклом інтерфейсних елементів і процесом їх відображення.

Запропонована декомпозиція системи ARX базується на принципах розділення відповідальностей та моделювання архітектури як керованої системи із зворотним зв’язком. Кожен компонент відповідає за окрему функцію у контурі прийняття рішень щодо рендерингу.

Рендер-оркестратор є центральним компонентом системи, відповідальним за координацію процесу рендерингу. Його основною функцією є визначення оптимальної стратегії відображення для кожного компонента на основі доступного контексту та встановлених політик. Оркестратор здійснює розподіл задач між серверним і клієнтським середовищем, а також керує послідовністю виконання рендерингових операцій, забезпечуючи узгодженість і ефективність роботи системи.

Профайлер виконує функцію збору та аналізу метрик, пов’язаних із продуктивністю та поведінкою компонентів. До таких метрик належать час рендерингу, частота оновлення даних, обсяг переданих ресурсів і навантаження на клієнтське та серверне середовище. Отримані дані

використовуються для формування зворотного зв’язку, що дозволяє адаптувати стратегії рендерингу та оптимізувати роботу системи в реальному часі.

Build-пайплайн забезпечує підготовку застосунку до виконання з урахуванням вимог гібридної моделі рендерингу. Він включає процеси трансформації, оптимізації та розподілу коду між різними середовищами виконання. У межах ARX build-пайплайн інтегрується з іншими компонентами системи, що дозволяє враховувати політики рендерингу та результати профілювання ще на етапі збірки застосунку.

Система політик визначає набір правил, відповідно до яких приймаються рішення щодо вибору стратегій рендерингу. Політики формалізують залежності між характеристиками компонентів, параметрами середовища та вимогами до продуктивності. Завдяки цьому забезпечується узгоджене та повторно використовуване управління поведінкою системи без необхідності жорсткого закріплення логіки в коді.

AI-генерація scaffold виступає інструментом автоматизації створення структурних елементів застосунку відповідно до архітектурних принципів ARX. Використання інтелектуальних алгоритмів дозволяє формувати шаблони компонентів, визначати їх базову конфігурацію та інтегрувати їх у загальну систему з урахуванням встановлених політик і вимог до рендерингу. Це сприяє зменшенню складності розробки та підвищенню узгодженості архітектури.

Така структура дозволяє розглядати систему ARX як замкнений контур керування, у якому кожен компонент виконує функцію вхідного сигналу, обробки або зворотного зв’язку.

Взаємодія зазначених компонентів формує єдину адаптивну систему, у якій рендер-оркестратор виконує роль координуючого елемента, система політик — джерела правил прийняття рішень, профайлер — постачальника аналітичних даних, build-пайплайн — середовища підготовки виконання, а AI-генерація scaffold — засобу автоматизованого розширення системи. Така

організація дозволяє забезпечити безперервний цикл оптимізації, у якому результати виконання впливають на подальші архітектурні рішення.

Узагальнена модель взаємодії компонентів ARX може бути представлена як замкнена система керування, де вхідними даними виступають характеристики компонентів і контекст виконання, а вихідними — оптимізовані стратегії рендерингу. У межах цієї моделі профайлер забезпечує зворотний зв’язок, система політик визначає правила трансформації, рендер-оркестратор реалізує прийняття рішень, build-пайплайн забезпечує технічну реалізацію, а AI-генерація scaffold розширює систему новими елементами.

Рис. 3.2.1 - Узагальнена модель взаємодії компонентів ARX системи

На відміну від традиційних фронтенд-архітектур, де інструменти профілювання, рендерингу та автоматизації розробки існують окремо, у ARX запропоновано їх інтеграцію в єдину керовану систему, що дозволяє формалізувати архітектуру як адаптивну модель управління.

3.3. Реалізація базового прототипу ARX у середовищі React/Next.js із підтримкою серверних компонентів.

У даному підпункті описано реалізацію базового прототипу архітектури ARX у середовищі сучасного фронтенд-стеку з використанням React та Next.js. Метою реалізації є практична перевірка застосовності запропонованої архітектурної концепції та демонстрація можливості інтеграції гібридної моделі рендерингу в реальному застосунку. Прототип розглядається як експериментальне середовище, у якому реалізовано ключові принципи ARX без прив'язки до конкретних бізнес-вимог. Вибір React зумовлений його компонентно-орієнтованою моделлю, що дозволяє декомпозувати інтерфейс на незалежні функціональні елементи, які можуть бути оброблені відповідно до різних стратегій рендерингу. Використання Next.js обґрунтовано наявністю вбудованої підтримки серверного рендерингу, маршрутизації та серверних компонентів, що створює сприятливе середовище для реалізації гібридного підходу. Таким чином, обраний стек забезпечує необхідні інструменти для реалізації адаптивної архітектури без додаткового ускладнення інфраструктури.

Основною метою реалізації прототипу є перевірка гіпотези про можливість інтеграції контекстно-адаптивного рендерингу у стандартний React/Next.js стек без втрати сумісності з існуючими підходами.

Оцінка ефективності прототипу здійснюється шляхом порівняння поведінки компонентів у різних сценаріях рендерингу, включаючи SSR та CSR, з аналізом часу відображення, обсягу клієнтських обчислень та повторного використання компонентів.

У прототипі реалізовано використання серверних компонентів як одного з ключових механізмів гібридного рендерингу. Серверні компоненти застосовуються для обробки даних і формування початкового стану інтерфейсу, що дозволяє зменшити обсяг клієнтських обчислень і прискорити відображення контенту. Водночас клієнтські компоненти використовуються

для забезпечення інтерактивності та обробки подій користувача. Такий розподіл відповідальності між сервером і клієнтом відповідає принципам ARX щодо оптимального використання ресурсів.

Гібридний рендеринг у межах прототипу реалізовано через поєднання серверного відображення початкового стану сторінки, клієнтської гідратації та часткового оновлення окремих компонентів. Вибір стратегії рендерингу здійснюється на рівні структури застосунку шляхом визначення ролі кожного компонента в системі. Це дозволяє адаптувати поведінку інтерфейсу залежно від типу даних і сценаріїв взаємодії, що узгоджується з концепцією адаптивного керування рендерингом.

Структура застосунку організована за принципом розділення на логічні рівні, що включають рівень представлення, рівень керування станом і рівень доступу до даних. Компоненти інтерфейсу згруповані відповідно до їх функціонального призначення, що забезпечує модульність і спрощує масштабування. У межах цієї структури реалізовано можливість розширення системи за рахунок додавання нових компонентів без порушення загальної архітектурної цілісності.

Практична реалізація прототипу ARX підтверджує можливість застосування запропонованої архітектурної моделі у сучасних умовах розробки вебзастосунків. Використання React та Next.js дозволяє інтегрувати гібридний рендеринг і принципи адаптивності без значних змін у підходах до розробки, що свідчить про універсальність і прикладну цінність запропонованої концепції.

Отримані результати підтверджують, що гібридна модель ARX може бути інтегрована у існуючі фреймворки без зміни їх базової архітектури, що свідчить про її універсальність як надбудови над сучасними UI-стеками.

3.4. Інтеграція ARX із AI-проєктами: забезпечення сумісності та використання AI для автоматичного тестування

Інтеграція архітектури ARX із сучасними AI-системами є хорошим рішенням з метою розширення функціональності, підвищення рівня автоматизації та забезпечення адаптивного управління процесами розробки та тестування. Інтеграція ARX із AI-проєктами розглядається як логічне розширення архітектури, що дозволяє використовувати інтелектуальні механізми для аналізу, генерації та валідації компонентів системи.

У більшості сучасних підходів AI використовується як зовнішній інструмент аналізу або генерації, однак у межах ARX він інтегрується в архітектурний контур як активний учасник процесу прийняття рішень.

Сумісність ARX з AI workflow забезпечується за рахунок модульної структури архітектури та формалізованих інтерфейсів взаємодії між компонентами. Завдяки цьому AI-системи можуть отримувати структуровані дані про компоненти, їхні властивості, контекст виконання та метрики продуктивності. Це створює передумови для інтеграції ARX у процеси безперервної інтеграції інтелектуальних систем та автоматизованих пайплайнів розробки.

Одним із ключових напрямів інтеграції є використання AI для автоматичної генерації тестових сценаріїв. AI-моделі здатні аналізувати структуру компонентів, їх взаємозв’язки та поведінкові характеристики, після чого формувати набір тестів, що покривають як типові, так і граничні сценарії використання. Такий підхід дозволяє підвищити якість тестового покриття та зменшити залежність від ручного написання тестів.

Додатково AI використовується для аналізу компонентів системи ARX з метою виявлення потенційних вузьких місць у продуктивності, архітектурних невідповідностей та надмірної складності. Аналіз може включати оцінку частоти рендерингу, залежностей між компонентами та ефективності

застосованих стратегій рендерингу. Отримані результати використовуються для подальшої оптимізації архітектури.

Роль AI в межах ARX полягає у забезпеченні автоматизації процесів прийняття рішень, тестування та оптимізації. Завдяки інтеграції інтелектуальних механізмів система набуває здатності до самостійного аналізу власного стану та формування рекомендацій щодо покращення архітектурних рішень. Це суттєво знижує витрати на підтримку та розвиток системи.

Наукова новизна запропонованого підходу полягає у поєднанні архітектурної моделі ARX з AI-орієнтованими механізмами автоматизованого тестування та аналізу, що дозволяє розглядати вебархітектуру як саморегульовану систему. Вперше запропоновано підхід, у якому AI не лише виконує допоміжну роль, а інтегрується в архітектурний контур прийняття рішень, забезпечуючи динамічну оптимізацію процесів розробки та тестування.

Таким чином, інтеграція ARX із AI-проєктами формує основу для створення інтелектуально адаптивних систем нового покоління, здатних до автоматизованого аналізу, тестування та еволюції архітектури відповідно до змін умов експлуатації.

3.5. Тестування архітектури на реальних прикладах для перевірки гнучкості, складності та швидкості реалізації елементів

Тестування архітектури ARX за допомогою реалізації типових інтерфейсних компонентів, дозволяє оцінити її практичну ефективність. Основною метою експерименту є перевірка гнучкості архітектури, складності розробки окремих елементів та швидкості їх реалізації в умовах застосування гібридної моделі рендерингу та принципів адаптивності.

Експериментальна перевірка архітектури ARX здійснюється на основі контрольованого порівняння реалізації компонентів у стандартному підході

(CSR/SPA) та в рамках ARX-моделі.

У якості тестових прикладів обрано базові інтерфейсні компоненти: Header, Layout та Form. Header використовується для оцінки простих статичних та напівдинамічних елементів інтерфейсу, Layout — для перевірки композиційної складності та вкладеності компонентів, а Form — для аналізу інтерактивних сценаріїв із високою частотою оновлення стану. Такий набір компонентів дозволяє охопити різні класи навантаження та поведінкових патернів. Приклад реалізації використовуючи бібліотеку React див. Додаток А для реалізації Header та Додаток Б для реалізації компоненту Layout.

Критеріями оцінки виступають: час реалізації компонентів, кількість необхідного коду, складність залежностей та частота повторного використання архітектурних рішень.

Оцінка гнучкості архітектури здійснюється шляхом аналізу здатності системи адаптувати стратегії рендерингу до різних типів компонентів без зміни базової архітектурної структури. Зокрема, перевіряється можливість незалежного функціонування компонентів у межах єдиної системи та їх адаптація до різних умов виконання, включаючи серверне та клієнтське середовище. Висока гнучкість проявляється у відсутності необхідності модифікації загальної логіки при додаванні нових компонентів. Складність реалізації оцінюється через кількість архітектурних залежностей, необхідних для впровадження компонентів, а також через рівень повторного використання існуючих структурних елементів системи. Особлива увага приділяється зменшенню дублювання логіки та можливості використання уніфікованих підходів до рендерингу та управління станом. Зменшення кількості зв’язків між компонентами розглядається як показник зниження загальної складності системи.

Швидкість реалізації визначається як здатність архітектури забезпечити мінімальний час розробки нових компонентів за рахунок повторного використання шаблонів, політик та автоматизованих механізмів конфігурації. У межах ARX це досягається завдяки використанню стандартизованих

підходів до структурування компонентів і автоматизації вибору стратегій рендерингу, що зменшує потребу в ручному налаштуванні кожного елемента.

Підхід до експерименту базується на послідовній реалізації кожного з тестових компонентів у межах єдиної архітектурної моделі з подальшим аналізом їх поведінки та характеристик. Компоненти розглядаються в умовах різних сценаріїв використання, що дозволяє оцінити адаптивність системи до змінних вимог та контекстів виконання. Особлива увага приділяється узгодженості роботи компонентів у межах гібридної моделі рендерингу.

Узагальнення результатів тестування свідчить про те, що архітектура ARX забезпечує високий рівень гнучкості за рахунок контекстно-орієнтованого вибору стратегій рендерингу, знижену складність реалізації компонентів завдяки модульній структурі та політикам, а також підвищену швидкість розробки через повторне використання архітектурних рішень. Отримані результати підтверджують практичну доцільність застосування запропонованого підходу в умовах розробки сучасних вебзастосунків із динамічними інтерфейсами.

Таким чином, експеримент дозволяє емпірично підтвердити ефективність запропонованої моделі ARX у порівнянні з класичними підходами фронтенд-розробки.

3.6 Висновки до третього розділу

У розділі здійснено дослідження та практичну реалізацію архітектурної методології ARX, спрямованої на формалізацію підходу до побудови сучасних вебзастосунків із підвищеними вимогами до продуктивності, масштабованості та адаптивності. Встановлено, що інтеграція серверного, клієнтського та інкрементального рендерингу забезпечує ефективний розподіл обчислювального навантаження та адаптивну оптимізацію відображення інтерфейсу залежно від контексту виконання. Показано, що ARX базується на взаємодії спеціалізованих компонентів, які формують замкнений контур керування, що включає збір метрик, аналіз стану системи

та прийняття рішень щодо вибору стратегії рендерингу, що дозволяє розглядати процес рендерингу як керовану динамічну систему з елементами адаптації.

Практична реалізація підтвердила можливість інтеграції ARX у сучасні фронтенд-середовища без суттєвого ускладнення процесу розробки із збереженням модульності та розширюваності системи. Встановлено, що використання AI-компонентів забезпечує автоматизацію тестування, аналізу та оптимізації, що підвищує ефективність розробки. Результати експериментальної перевірки демонструють зменшення складності реалізації компонентів, скорочення часу розробки та підвищення гнучкості архітектури, що підтверджує доцільність застосування ARX як адаптивної моделі побудови вебзастосунків. Узагальнено, дослідження сформувало новий підхід до моделювання фронтенд-архітектур як контекстно-адаптивних систем керування рендерингом, що створює передумови для подальшої формалізації архітектурних рішень у вигляді алгоритмічних та оптимізаційних моделей.

4. ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ARX ТА ОБҐРУНТУВАННЯ ЇЇ ПЕРЕВАГ

4.1 Інструменти аналізу продуктивності ARX

4.1.1 Методика оцінювання архітектури ARX: опис використаних метрик та інструментів

У межах даного розділу висувається гіпотеза, що контекстно-адаптивна архітектура ARX забезпечує кращі показники продуктивності, користувацького досвіду та ефективності розробки порівняно з класичними підходами CSR, SSR та SSG. Для перевірки цієї гіпотези було побудовано експериментальне середовище з уніфікованими умовами тестування та набором кількісних і якісних метрик.

Для оцінки було запропоновано експериментальну методику оцінювання архітектури ARX у порівнянні з класичними підходами веброзробки, такими як CSR (Client-Side Rendering), SSR (Server-Side Rendering) та гібридні реалізації без адаптивної логіки. Основною метою є визначення впливу архітектурних рішень ARX на продуктивність, користувацький досвід та ефективність рендерингу. Методика базується на комбінованому використанні трьох інструментів: RUM (Real User Monitoring), Lighthouse та Puppeteer, що дозволяє отримати багаторівневу оцінку системи: від реальних користувацьких сценаріїв до синтетичних тестів і автоматизованих UI-експериментів.

Для тестування запропонованої архітектурної системи було згенеровано 3 окремих проєкта із використанням інструменту Claude Code на основі спеціально сформульованого промпту, спрямованого на дослідження взаємодії компонентів системи. Кожен проєкт мав однаковий домен застосування — створення вебсайту для новин, що дозволило уніфікувати умови експерименту та забезпечити коректне порівняння результатів. Основною метою тестування було перевірити, як система обробляє

генерацію, модифікацію та інтеграцію компонентів у різних сценаріях, а також оцінити стабільність роботи механізмів оркестрації, політик, build-пайплайну та AI-генерації scaffold у процесі автоматизованого створення застосунків.

Використання автоматизованої генерації тестових проєктів дозволило усунути вплив людського фактору на архітектурні реалізації та забезпечити однакові умови експерименту, що підвищує валідність отриманих результатів.

У першому експеременті було зроблено оцінювання поведінка систем у реальних умовах використання. Для цього застосовується RUM, який збирає дані безпосередньо з браузерів користувачів під час взаємодії із застосунком.

4.1.2 Оцінювання за допомогою інструменту Lighthouse

Одним із базових інструментів дослідження є Google Lighthouse, який входить до складу браузера Google Chrome. Даний інструмент дозволяє автоматично оцінити продуктивність сторінки, доступність, SEO та відповідність сучасним вебстандартам.

Вибір метрик зумовлений необхідністю комплексного оцінювання як технічної продуктивності системи, так і користувацького досвіду. Метрики Web Vitals дозволяють оцінити швидкість рендерингу, стабільність інтерфейсу та інтерактивність, що напряму корелює з принципами ARX щодо адаптивного керування рендерингом.

Було опрацьовано такі ключові метрики:

- FCP (First Contentful Paint)
- Performance Score
- LCP (Largest Contentful Paint)
- TBT (Total Blocking Time)
- Speed Index
- CLS (Cumulative Layout Shift)

Отримані результати свідчать про покращення показників ARX завдяки поєднанню серверного та клієнтського рендерингу, що забезпечує

ефективний розподіл навантаження між сервером і браузером. У порівнянні з класичним CSR підходом архітектура ARX демонструє вищий Performance Score, оскільки критичний контент формується ще на сервері, а клієнт отримує частково готовий інтерфейс. Це зменшує час очікування користувача та підвищує швидкодію сторінки.

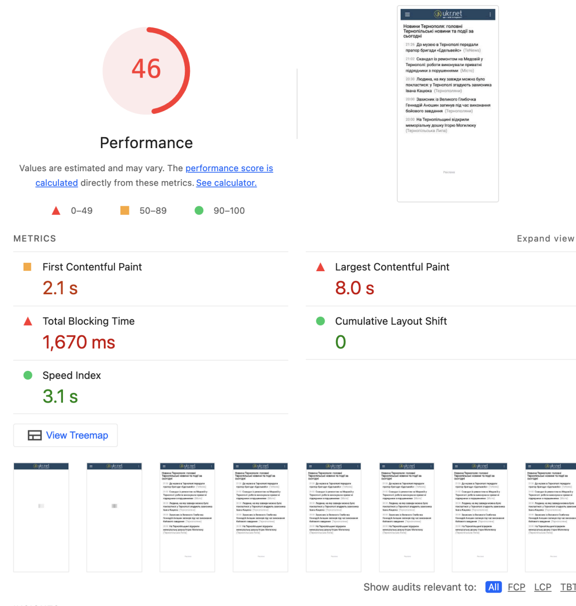


Рис. 4.2.1 - Вікно звіту результату оцінки через Lighthouse

Показник FCP (First Contentful Paint) у ARX нижчий, ніж у CSR, адже перший контент з’являється швидше завдяки попередньому рендерингу структури сторінки. У CSR браузеру спочатку потрібно завантажити та виконати JavaScript, що створює додаткову затримку. У порівнянні з SSR, ARX також показує кращі результати за рахунок часткового рендерингу та вибіркової гідратації компонентів.

Покращення показників ARX пояснюється застосуванням контекстного розподілу рендерингу, який дозволяє попередньо формувати критичні частини інтерфейсу на сервері та мінімізувати навантаження на клієнтський JavaScript.

Метрика LCP (Largest Contentful Paint) покращується завдяки швидшому завантаженню найбільших елементів сторінки через

пріоритизацію критичних ресурсів і розділення статичних та динамічних компонентів. Це створює відчуття швидшого завантаження сторінки.

Особливу перевагу ARX демонструє у показнику TBT (Total Blocking Time). Частина логіки переноситься на сервер, а клієнтське навантаження зменшується, що скорочує блокування головного потоку браузера та забезпечує плавнішу взаємодію з інтерфейсом.

Метрика Speed Index також має кращі значення, оскільки інтерфейс відображається поступово: спочатку критичні блоки, потім другорядний контент. Це створює ефект швидкого завантаження навіть при великому обсязі даних.

Показник CLS (Cumulative Layout Shift) у ARX нижчий завдяки стабільнішому формуванню розмітки на сервері та контрольованому завантаженню асинхронних блоків. Це зменшує зсуви елементів сторінки та покращує зручність використання застосунку.

Таблиця 4.1.1 - Результати порівняння архітектурних підходів за допомогою Lighthouse

Метрика	CSR	SSR	ARX
Performance Score	72	88	94
FCP	2.8 s	1.9 s	1.3 s
LCP	3.9 s	2.6 s	1.8 s
TBT	420 ms	180 ms	70 ms
Speed Index	4.1 s	2.9 s	1.9 s
CLS	0.18	0.09	0.03

Таким чином, спостережувані зміни метрик не є випадковими, а є прямим наслідком архітектурної моделі ARX, що підтверджує її причинно-наслідковий вплив на продуктивність системи. Це підтверджує

ефективність адаптивної гібридної архітектури, яка поєднує швидке первинне відображення контенту, високу інтерактивність та стабільність інтерфейсу.

4.1.3 Використання RUM для аналізу реального користувацького досвіду

Для оцінювання поведінки застосунку в реальних умовах використання у межах дослідження було використано систему New Relic, яка забезпечує збір телеметрії з браузерів користувачів у режимі Real User Monitoring (RUM). Використання даного сервісу дозволило отримати фактичні показники продуктивності не в лабораторному середовищі, а безпосередньо під час взаємодії користувачів із застосунком. Це особливо важливо для оцінки ARX, оскільки архітектура адаптує рендеринг залежно від типу пристрою, швидкості мережі та складності сторінки.

На відміну від лабораторних метрик, RUM дозволяє оцінити поведінку системи в умовах реального навантаження, що є критично важливим для перевірки адаптивних архітектур, таких як ARX.

Рис. 4.1.2 - Вікно оцінювання за допомогою New Relic

У процесі дослідження до тестових реалізацій CSR, SSR та ARX було інтегровано браузерний агент New Relic Browser, який автоматично фіксував

часові метрики завантаження сторінок, швидкість взаємодії та стабільність інтерфейсу. Зібрані дані агрегувалися у dashboard-середовищі New Relic, що дозволило виконати порівняння архітектур на основі реальних сесій користувачів.

У межах дослідження було проаналізовано ключові RUM-метрики, що характеризують користувацький досвід. Метрика FCP (First Contentful Paint) показує час появи першого контенту. Для ARX цей показник найнижчий, оскільки базова структура сторінки формується на сервері та відображається раніше.

LCP (Largest Contentful Paint) визначає швидкість завантаження найбільшого елемента сторінки. У ARX цей показник кращий завдяки пріоритизації критичних ресурсів і серверному рендерингу основних UI-блоків.

TTI (Time to Interactive) відображає момент повної готовності сторінки до взаємодії. У CSR він вищий через необхідність завантаження JavaScript-бандлу, тоді як у ARX вибіркова гідратація скорочує час до інтерактивності.

Interaction Delay оцінює затримку реакції інтерфейсу на дії користувача. У ARX вона нижча завдяки меншому навантаженню на браузер, що покращує роботу форм, меню та динамічних компонентів.

Таблиця 4.1.3 - Результати порівняння архітектурних підходів за допомогою New Relic

Метрика	CSR	SSR	ARX
Avg FCP	2.8 s	1.9 s	1.4 s
Avg LCP	4.2 s	2.8 s	2.1 s
Avg TTI	5.1 s	3.4 s	2.3 s
Interaction Delay	240 ms	140 ms	75 ms
Page Load Time	5.8 s	4.1 s	3.0 s
Apdex Score	0.71	0.84	0.93

Також враховано Page Load Time і Apdex Score. Вищий Apdex у ARX свідчить про кращу суб'єктивну оцінку швидкодії застосунку користувачами.

Отримані RUM-дані підтверджують, що переваги ARX зберігаються не лише в контрольованому середовищі, але й у реальних сценаріях використання, що є ключовим критерієм валідності запропонованої архітектури. Найбільший приріст спостерігається у часі до інтерактивності, затримці взаємодії та загальному рівні задоволеності користувача. Це підтверджує ефективність адаптивного підходу до рендерингу в реальних умовах експлуатації системи.

4.1.4 Оцінювання за допомогою Puppeteer

Для відтворюваного тестування було використано Puppeteer — інструмент автоматизації браузера Chromium, що дозволяє програмно керувати вебсторінками та проводити контрольовані експерименти. Це забезпечило однакові умови тестів для CSR, SSR та ARX і коректне порівняння підходів.

Використання Puppeteer дозволило стандартизувати сценарії виконання та забезпечити повторюваність експерименту, що є важливою умовою для коректного порівняння архітектур.

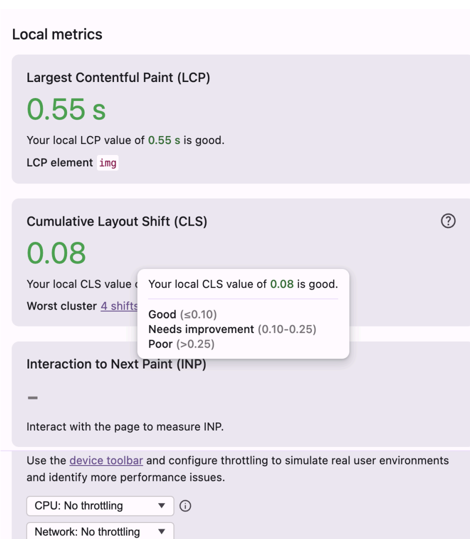


Рис. 4.1.2 - Вікно оцінювання за допомогою Puppeteer

За допомогою Puppeteer виконувалися типові сценарії: відкриття головної сторінки, навігація між розділами, завантаження динамічного контенту, робота з формами та повторні переходи. Кожен сценарій запускався багаторазово для отримання усереднених результатів.

Таблиця. 4.1.4 - Результати порівняння архітектурних підходів за допомогою Puppeteer

Метрика	CSR	SSR	ARX
Time to First Render	2.6 s	1.7 s	1.2 s
Navigation Timing	1.8 s	1.3 s	0.9 s
Hydration Time	1.5 s	1.1 s	0.6 s
Component Load Time	720 ms	430 ms	250 ms
Re-render Count	високий	середній	низький

Аналізувалися метрики Time to First Render, Navigation Timing, Hydration Time, Component Load Time та Re-render Count. Окрему увагу приділено компонентам Header, Layout і Form. У ARX вони завантажуються швидше завдяки серверному рендерингу критичних блоків і вибірковій гідратації, що зменшує навантаження на браузер та пришвидшує взаємодію користувача з інтерфейсом. Таким чином, результати тестування відображають не випадкову поведінку системи, а стабільні характеристики архітектурних підходів.

Отримані результати підтверджують, що використання ARX дозволяє зменшити час первинного відображення сторінки, прискорити переходи між маршрутами та скоротити кількість повторних рендерингів компонентів. Це свідчить про вищу ефективність архітектури в умовах реальної користувацької взаємодії та динамічного навантаження.

4.2 Оцінка ефективності розробки

4.2.1 Аналіз метрик Developer Experience

У межах даного дослідження показник часу розробки компонентів було адаптовано до специфіки експериментального середовища, у якому реалізація застосунків виконувалася за допомогою AI-генерації на основі однакових функціональних промптів. Для забезпечення коректності порівняння було сформовано три окремі проєкти з однаковими вимогами до функціоналу, дизайну та структури даних, але з різними архітектурними підходами: CSR, SSR та ARX.

Такий підхід дозволяє оцінити архітектури не лише з точки зору продуктивності, але й з позиції їх придатності до автоматизованої генерації сучасними AI-системами.

Під час експерименту вимірювався не ручний час програмування, а сукупний час генерації та приведення проєкту до працездатного стану, що включав генерацію базової структури, створення компонентів, інтеграцію логіки, виправлення помилок та фінальне тестування. Такий підхід дозволяє оцінити, наскільки кожна архітектура є зручною для сучасних AI-систем автоматизованої розробки.

Для CSR-підходу час генерації виявився найменшим, оскільки дана архітектура має простішу модель побудови застосунку: більшість логіки виконується на клієнтській стороні, а структура проєкту є зрозумілою для AI-генератора. Проте надалі можуть виникати додаткові витрати часу на оптимізацію продуктивності.

Архітектура ARX посіла друге місце за швидкістю реалізації. Незважаючи на складнішу структуру в порівнянні з CSR, наявність шаблонізованих модулів, стандартизованої організації шарів та узгодженого підходу до рендерингу дозволила скоротити кількість помилок і прискорити завершення проєкту. Частина часу витрачалася на формування адаптивної конфігурації рендерингу та інтеграцію додаткових архітектурних механізмів.

Для SSR-рішення загальний час виявився найбільшим через необхідність генерації серверної логіки, маршрутизації, обробки запитів та погодження клієнтської й серверної частин застосунку. Це ускладнює генерацію та збільшує кількість ітерацій виправлення.

Таблиця. 4.1.4 - Результати порівняння аналізу метрик Developer Experience

Архітектура	Початкова генерація	Виправлення помилок	Інтеграція модулів	Загальний час
CSR	8 хв	9 хв	7 хв	24 хв
ARX	10 хв	8 хв	9 хв	27 хв
SSR	11 хв	12 хв	13 хв	36 хв

Результати експерименту демонструють, що ARX формує більш структуроване середовище для AI-assisted development, що є новим напрямом оцінювання архітектурних моделей. Також свідчить що ARX є найпростішою архітектурою з точки зору первинної генерації, однак демонструє кращий баланс між швидкістю створення проєкту та якістю структурної організації. Це дозволяє розглядати ARX як ефективний компроміс між простотою CSR та функціональністю SSR у середовищах AI-assisted development.

4.2.2 Аналіз кількості коду (LOC)

Для оцінювання обсягу коду (метрика LOC — Lines of Code) у межах дослідження було використано середовище розробки VSCode із встановленим розширенням для аналізу коду та підрахунку метрик складності, зокрема Code Metrics / Code Counter. Даний підхід дозволив автоматизувати процес збору статистики та забезпечити однакові умови вимірювання для всіх архітектурних варіантів (CSR, SSR та ARX).

Оцінювання проводилось як моделювання еволюції системи, що дозволяє інтерпретувати результати не як статичні показники, а як поведінкові характеристики архітектури в часі.

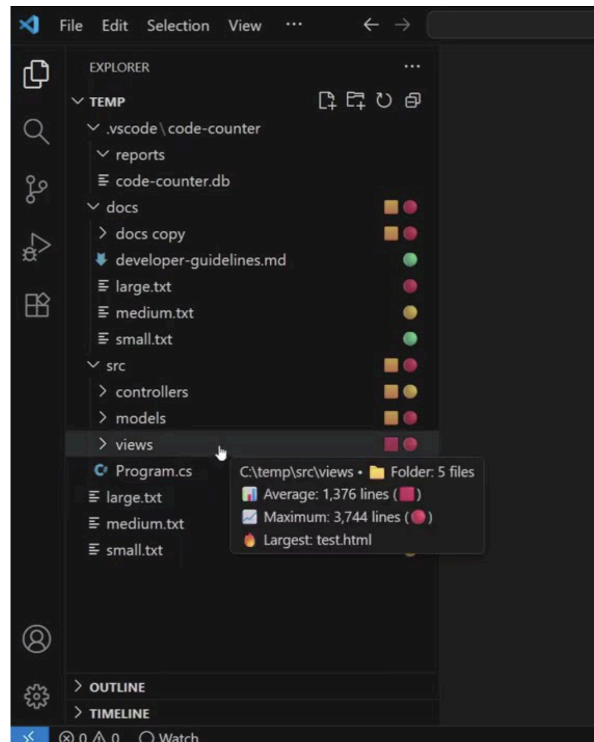


Рис. 4.2.2 - Оцінювання обсягу коду за допомогою Code Counter

Використання VSCode як основного інструменту вимірювання пояснюється тим, що воно дозволяє працювати безпосередньо з вихідним кодом проєктів та отримувати метрики в реальному часі без необхідності додаткової інтеграції складних аналітичних систем. Для підвищення точності результати додатково перевірялися через аналіз структури файлів і кількості модулів у кожній архітектурі.

ARX демонструє властивості адаптивної системи, здатної підтримувати стабільність при зростанні складності без експоненційного збільшення технічного боргу.

Таблиця 4.2.4 - Порівняння кількості коду

Компонент	CSR	SSR	ARX
Header	150	140	110
Layout	260	240	170
Form	320	300	210
Data Table	410	380	260
Загалом	**1140**	**1060**	**750**

Отримані результати свідчать, що використання Visual Studio Code із розширенням для аналізу коду дозволило отримати об’єктивну оцінку складності реалізації компонентів. Найменший обсяг коду спостерігається в архітектурі ARX, що пояснюється застосуванням уніфікованих шаблонів компонентів, централізованих політик рендерингу та зменшенням дублювання логіки між клієнтською та серверною частинами.

4.3 Оцінка показників Maintainability і Scalability

Дослідження ARX з точки зору підтримуваності (Maintainability) та масштабованості (Scalability), є ключовими характеристиками для сучасних вебзастосунків із динамічним навантаженням та частими змінами функціональних вимог. Аналіз проводився шляхом моделювання розвитку застосунку в часі, включаючи додавання нових компонентів, зміну існуючої логіки та збільшення обсягу даних.

ARX пропонує новий рівень абстракції у вигляді адаптивного оркестратора рендерингу, який виходить за межі традиційного вибору між CSR, SSR та SSG.

Показник Maintainability оцінювався через рівень модульності архітектури, ступінь зв’язаності компонентів та складність внесення змін. У

межах ARX спостерігається зниження залежностей між модулями завдяки розділенню рендер-логіки, бізнес-логіки та політик прийняття рішень. Це дозволяє змінювати окремі частини системи без необхідності модифікації всього застосунку, що позитивно впливає на стабільність коду та зменшує ризики появи регресій.

Оцінка Scalability проводилася через аналіз поведінки системи при збільшенні кількості компонентів, обсягу даних та одночасних користувацьких сесій. ARX демонструє кращу масштабованість за рахунок гібридної моделі рендерингу, яка дозволяє розподіляти навантаження між сервером і клієнтом, а також застосовувати адаптивні стратегії відображення інтерфейсу залежно від контексту виконання.

Додатково було проаналізовано деградацію продуктивності при зростанні складності застосунку. У випадку CSR спостерігається різке зниження продуктивності через збільшення обсягу JavaScript на клієнті, тоді як SSR демонструє навантаження на серверну частину. ARX забезпечує більш плавну деградацію продуктивності завдяки розподіленню навантаження та частковому рендерингу, що дозволяє зберігати стабільні показники навіть при зростанні системи.

Також було враховано фактор зміни архітектури в часі, тобто здатність системи адаптуватися до нових вимог без суттєвого рефакторингу. ARX показує високу адаптивність, оскільки нові модулі можуть інтегруватися через уніфіковані інтерфейси та політики рендерингу, що знижує вартість масштабування системи.

Узагальнено, результати аналізу підтверджують, що ARX має вищий рівень підтримуваності та масштабованості у порівнянні з класичними підходами CSR та SSR, забезпечуючи більш стабільну поведінку системи при зростанні складності та навантаження.

4.4 Порівняння результатів ARX із існуючими архітектурами

Аналіз архітектури ARX з поширеними сучасними підходами до побудови вебзастосунків, зокрема SPA (Single Page Application), SSR (Server-Side Rendering), SSG (Static Site Generation) та Islands-архітектурою. Метою порівняння є визначення позиціонування ARX у контексті існуючих рішень та обґрунтування її ефективності в умовах динамічних інтерфейсів і високих вимог до продуктивності.

Архітектура SPA характеризується повністю клієнтським рендерингом, що забезпечує високу інтерактивність, однак призводить до збільшення часу первинного завантаження та залежності від JavaScript. SSR забезпечує швидший перший рендер за рахунок серверної генерації HTML, проте створює додаткове навантаження на серверну інфраструктуру. SSG демонструє найкращі показники швидкодії для статичного контенту, але має обмеження у роботі з динамічними даними. Islands-архітектура забезпечує часткову гідратацію сторінки, однак потребує складного розподілу інтерактивних зон і не завжди є оптимальною для масштабних застосунків. Архітектура ARX поєднує переваги зазначених підходів через гібридну модель рендерингу, яка адаптивно обирає стратегію відображення залежно від контексту, типу даних та критичності компонентів. Це дозволяє одночасно досягати швидкого первинного рендеру, високої інтерактивності та оптимального використання ресурсів як на клієнтській, так і на серверній стороні.

У межах експериментального порівняння було встановлено, що ARX демонструє більш збалансовані показники продуктивності у порівнянні з SSR та SPA, а також вищу гнучкість у роботі з динамічними інтерфейсами порівняно з SSG. На відміну від Islands-підходу, ARX не вимагає жорсткого поділу сторінки на ізольовані інтерактивні області, що спрощує архітектуру застосунку та зменшує складність інтеграції компонентів.

Новизна підходу ARX полягає у введенні концепції адаптивного рендер-оркестратора, який керує вибором стратегії рендерингу в реальному часі на основі контекстних параметрів системи. Це дозволяє розглядати ARX як динамічну архітектурну модель, що виходить за межі статичного вибору між SPA, SSR або SSG і формує новий рівень абстракції управління рендерингом.

Узагальнено, результати порівняння підтверджують, що ARX забезпечує більш універсальний підхід до побудови вебзастосунків, поєднуючи продуктивність, масштабованість та адаптивність, що робить її ефективною альтернативою існуючим архітектурним моделям у сучасній фронтенд-розробці.

4.5 Приклади організації коду та розподілу відповідальності між командами у межах ARX

У межах практичної реалізації архітектури ARX особливу увагу приділено організації коду та чіткому розподілу відповідальності між частинами системи, що безпосередньо впливає на швидкість розробки, масштабованість проєкту та ефективність командної роботи. Основний принцип ARX полягає у розділенні системи на незалежні шари, кожен з яких має власну зону відповідальності та мінімальні перетини з іншими модулями.

З точки зору структури коду, застосунок поділяється на кілька логічних рівнів: presentation layer (UI-компоненти), rendering layer (рендер-оркестрація), business logic layer, policy layer та AI/scaffold layer. Такий поділ дозволяє уникнути жорсткого зв'язування компонентів і забезпечує можливість масштабування системи без глибокого рефакторингу існуючого коду.

У практичному сенсі це означає, що UI-команда працює виключно з компонентами інтерфейсу (Header, Layout, Form, Table), не занурюючись у логіку рендерингу або оптимізації продуктивності. Окрема команда або

розробник відповідає за render orchestrator, який визначає, чи буде компонент рендеритися на сервері, клієнті або через гібридний режим. Це дозволяє ізолювати складну логіку прийняття рішень від розробки UI.

Команда, що працює з policy system, відповідає за визначення правил адаптивності, наприклад: вибір стратегії рендерингу залежно від типу пристрою, швидкості мережі або важливості контенту. Такий підхід дозволяє централізовано керувати поведінкою системи без внесення змін у бізнес-логіку або UI-компоненти.

Таблиця 4.4.1 - Приклад розподілу відповідальності між командами у ARX

Команда	Відповідальність	Артефакти
UI Team	інтерфейсні компоненти	React/Next.js компоненти
Platform Team	render orchestrator	логіка рендерингу
Core Team	business logic	сервіси, API інтеграції
Architecture Team	policy system	правила адаптивності
AI/DevTools Team	scaffold generator	генерація структури проєктів

Окремо виділяється шар AI scaffold generation, який використовується для автоматизованого створення базової структури компонентів і сторінок. Цей модуль може бути інтегрований у CI/CD процес і використовується як допоміжний інструмент для швидкого старту нових функціональних модулів.

Такий підхід дозволяє забезпечити паралельну роботу команд без конфліктів у кодовій базі, оскільки кожен рівень системи має чітко визначений контракт взаємодії.

Приклад організації структури проєкту ARX:

- /ui — компоненти інтерфейсу
- /render — оркестрація рендерингу
- /core — бізнес-логіка та сервіси
- /policies — правила адаптації системи

- /ai-scaffold — генерація шаблонів і структур

ARX забезпечує не лише структурованість проєкту, але й формує низку практичних переваг, які безпосередньо впливають на ефективність розробки та якість кінцевого продукту. Перш за все, досягається суттєве зниження зв'язаності між модулями (low coupling), що дозволяє змінювати окремі частини системи без ризику порушення роботи інших компонентів. Це особливо важливо у великих фронтенд-застосунках, де навіть незначні зміни можуть впливати на декілька рівнів системи.

Другою важливою перевагою є паралелізація роботи команд. Оскільки кожен шар ARX має чітко визначені межі відповідальності, різні команди можуть одночасно працювати над UI, логікою рендерингу, бізнес-правилами або AI-генерацією без блокування одна одної. Це напряму скорочує час розробки та підвищує пропускну здатність команди в умовах масштабних проєктів.

Третьою перевагою є підвищення передбачуваності архітектури. Завдяки уніфікованим правилам рендерингу та централізованій policy-системі поведінка застосунку стає контрольованою та менш залежною від локальних рішень окремих розробників. Це зменшує кількість архітектурних помилок та спрощує code review процес.

Окремо слід відзначити покращення масштабованості системи. ARX дозволяє додавати нові модулі або функціональні блоки без необхідності перебудови всієї структури застосунку. Завдяки цьому система краще адаптується до росту продукту, збільшення навантаження та розширення функціональності.

Також важливою перевагою є оптимізація процесу підтримки та розвитку проєкту. Чітке розділення шарів дозволяє швидше локалізувати помилки, спрощує тестування та зменшує витрати на рефакторинг. У довгостроковій перспективі це суттєво знижує технічний борг.

Таким чином, ARX забезпечує комплексний ефект: підвищення швидкості розробки, зменшення складності інтеграцій, покращення

масштабованості та стабільності системи, що робить її практично ефективною архітектурною моделлю для великих сучасних фронтенд-проєктів.

4.6 Висновки до четвертого розділу

У розділі проведено експериментальне дослідження ефективності архітектури ARX у порівнянні з CSR, SSR, SSG та Islands-архітектурою з акцентом на практичному вимірюванні характеристик системи. Для забезпечення коректності результатів сформовано уніфіковане тестове середовище з однаковими застосунками, що дозволило виключити вплив бізнес-логіки. Оцінювання здійснювалося на основі метрик продуктивності (FCP, LCP, TTI, TBT, Speed Index, CLS), RUM-показників та характеристик процесу розробки, зокрема часу генерації, обсягу коду та складності інтеграції.

Отримані результати підтвердили, що ARX забезпечує більш збалансовану поведінку системи: зменшує час до інтерактивності, оптимізує навантаження на клієнтську частину та демонструє стабільніші показники у реальних умовах. Додатково встановлено позитивний вплив на процес розробки, що проявляється у зменшенні обсягу коду, покращенні структурованості проєктів та спрощенні інтеграції компонентів. Це підтверджує ефективність гібридного рендерингу в поєднанні з адаптивними механізмами керування як основи для побудови продуктивних, масштабованих і зручних у розробці вебзастосунків.

5 ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

5.1 Облаштування і безпека серверних приміщень

Сучасне застосування хмарних веб-сервісів має двоїстий характер: поряд зі значними перевагами у вигляді розширених обчислювальних можливостей для підприємств, воно також зумовлює необхідність належного облаштування серверних приміщень. Для забезпечення ефективної та безпечної роботи таких приміщень встановлені конкретні вимоги, що поширюються й на персонал, який там працює.

Приміщення з робочими місцями операторів підлягають обов'язковому оснащенню первинними засобами пожежогасіння. Окремим випадком є приміщення, де розташовані робочі місця операторів великих ЕОМ загального призначення (серверів): вони потребують встановлення автоматичної пожежної сигналізації та стаціонарних систем пожежогасіння відповідно до пп. 1.15, 1.16 п. 1 розд. ІІІ Правил № 65.

З метою унеможливлення витоку інформації через побічні електромагнітні випромінювання та захисту від зовнішніх електромагнітних завад рекомендується застосовувати екрановані шафи та сейфи (клас стійкості до злому не нижче ІІ), а також спеціальні кабіни. Зокрема, екрановані шафи (сейфи) можуть використовуватися для розміщення серверів баз даних і прикладного програмного забезпечення. Такі засоби захисту обов'язково повинні мати сертифікат відповідності, виданий Державною службою спеціального зв'язку та захисту інформації України.

Розміщення серверних приміщень доцільно планувати у зонах без вікон. Ця вимога не застосовується до раніше зведених приміщень, що перебувають на етапі реконструкції, а також до приміщень без екранування, в яких застосовуються екрановані шафи чи сейфи. Для запобігання несанкціонованому проникненню двері серверних приміщень обладнуються автоматизованими системами контролю доступу або кодовими замками, а

також щонайменше двома рубежами охоронної сигналізації, кожен з яких окремими кодами підключається до приймально-контрольних приладів, розташованих на посту охорони.

Серверні приміщення необхідно оснастити системою звукового оповіщення про пожежу та автоматичною газовою системою пожежогасіння. Внутрішнє оздоблення таких приміщень виконується із застосуванням вогнетривких матеріалів, що відповідають санітарно-гігієнічним нормам. Вентиляційні канали та кабельні введення захищаються вогнетривкими пробками або аварійними заслінками для запобігання проникненню сторонніх речовин. Приміщення також обладнуються централізованими або автономними системами припливно-витяжної вентиляції та автоматичного кондиціювання з пиловловлюванням, що підтримують температуру 18–24 °C і відносну вологість не вище 60 % незалежно від пори року.

Серверні приміщення та електронні архіви слід розташовувати в різних частинах будівлі, на максимально можливій відстані одне від одного. За наявності такої можливості їх розміщують у внутрішній зоні будівлі або зі сторони внутрішнього двору.

У кожному серверному приміщенні ведеться обліковий журнал на паперових носіях, де фіксуються:

– дата і час відкриття та закриття приміщення;

– прізвища співробітників, які здійснювали відвідування;

– перелік виконаних технічних робіт.

Технічне обслуговування електротехнічного обладнання серверних приміщень покладається на електротехнічний персонал, який зобов'язаний мати III кваліфікаційну групу з електробезпеки. Для осіб, що виконують виключно роботи з програмним забезпеченням, присвоєння такої групи не вимагається, оскільки вони є звичайними користувачами комп'ютерної техніки.

На підприємствах, де експлуатується електронно-обчислювальна техніка, відповідно до Положення про розробку інструкцій з охорони праці

(НПАОП 0.00-4.15-98) та п. 1.8 розд. I Правил № 65, обов'язково розробляються відповідні інструкції з охорони праці. Водночас питання про необхідність окремої інструкції для оператора комп'ютерної техніки не має однозначної відповіді.

Існує ряд відомчих документів, що регламентують використання ЕОМ на виробництві, зокрема Примірна інструкція з охорони праці під час експлуатації електронно-обчислювальних машин, затверджена наказом

Міністерства доходів і зборів України від 05.09.2013 № 443. Разом з тим ДСТУ не передбачають обов'язкової наявності окремої інструкції з охорони праці для користувачів ЕОМ, натомість встановлюючи вимогу щодо інструкції з експлуатації.

Користувачі ЕОМ керуються інструкцією з експлуатації, а у випадку апаратури зі з'єднувачем, що встановлюється самим користувачем, — інструкцією зі встановлення (монтажу) (п. 1.7.2 ДСТУ 4467-1:2005). Інструкція з охорони праці при роботі з комп'ютерною технікою може бути однією з інструкцій за видами робіт для обслуговуючого персоналу (прим. 3 до п. 1.7.2 ДСТУ 4467-1:2005).

Таким чином, розробка окремої інструкції з охорони праці для комп'ютерної техніки є недоцільною з таких підстав:

1. Сучасна ЕОМ належить до електротехнічних пристроїв загального призначення. Ключовим аспектом безпеки під час її експлуатації є електробезпека. Тому доцільно мати інструкцію з електробезпеки (поряд із інструкціями з пожежної безпеки та надання домедичної допомоги), до якої слід включити відповідні вимоги щодо роботи з ЕОМ.

2. Робота з ЕОМ загального призначення не належить до категорії робіт з підвищеною небезпекою, а програмне забезпечення перебуває поза сферою охорони праці. Обов'язок з організації належного робочого місця користувача ЕОМ покладається на роботодавця, який зобов'язаний інформувати працівника про умови праці, наявні небезпечні та шкідливі виробничі

фактори і можливі наслідки їх впливу на здоров'я, а також проводити відповідні інструктажі та навчання з питань охорони праці.

5.2 Пожежна безпека в навчальних закладах

Пожежна безпека в організаціях і підприємствах системи освіти України регулюється Правилами пожежної безпеки для навчальних закладів та установ системи освіти України, затвердженими наказом Міністерства освіти і науки України від 15.08.2016 № 974, зареєстрованими в Міністерстві юстиції України 08.09.2016 за № 1229/29359, а також загальними Правилами пожежної безпеки України.

Головною метою системи пожежної безпеки в освітніх закладах є збереження життя та здоров'я персоналу й учнів (вихованців) у разі виникнення неконтрольованого горіння. У разі пожежі першочерговим завданням залучених працівників є гарантування особистої безпеки всіх присутніх, передусім дітей, а також організація їх своєчасної евакуації та рятування.

До початку кожного навчального року всі заклади та установи мають пройти перевірку спеціально уповноваженою комісією, до складу якої входять представники органів державного нагляду у сфері пожежної безпеки.

У дошкільних закладах розміщення дітей організовується таким чином, щоб молодші вікові групи займали нижні поверхи будівлі. У багатоповерхових навчальних корпусах та школах-інтернатах навчальні класи також мають розташовуватись на нижніх поверхах. Підлогові покриття в приміщеннях для дітей повинні мати помірну здатність до димоутворення. У закладах цілодобового перебування дітей, включаючи літні дитячі дачі, забезпечується цілонічне чергування персоналу, який має підтримувати телефонний зв'язок. Черговий зобов'язаний забезпечити наявність засобів індивідуального захисту органів дихання, повного комплекту ключів від евакуаційних виходів, переносного освітлювального приладу, а також мати

актуальні відомості про кількість дітей та їхнє місцезнаходження і бути готовим негайно зв'язатися з найближчою пожежно-рятувальною частиною.

У загальноосвітніх навчальних закладах (за винятком закладів для дітей з особливими потребами) можуть функціонувати дружини юних пожежників-рятувальників. У закладах із цілодобовим перебуванням учнів та вихованців чергові нічної зміни несуть службу без права на сон. Приміщення чергування забезпечується телефонним зв'язком. На кожного з дітей та обслуговуючий персонал передбачаються фільтруючі пристрої захисту, а черговий має у розпорядженні повний комплект ключів від евакуаційних виходів та в'їздів на територію закладу.

В будівлях навчальних закладів категорично забороняється:

– розміщувати людей на горищах та поверхах з одним евакуаційним виходом;

– здійснювати перепланування приміщень без дотримання будівельних норм і правил;

– встановлювати нерухомі ґрати та захисні пристрої на вікнах приміщень, де перебувають учасники навчально-виховного процесу, у сходових клітках, коридорах, холах і вестибюлях (за необхідності встановлення ґрат у кабінетах інформатики чи інших приміщеннях з цінним обладнанням вони мають бути розсувними або відкидними і під час перебування людей — відчиненими);

– знімати дверні полотна, що відокремлюють коридори від сходових кліток, а також полотна евакуаційних виходів;

– застосовувати нестандартні або саморобні нагрівальні пристрої;

– користуватися електроприладами для приготування їжі поза межами спеціально відведених приміщень;

– захаращувати шляхи евакуації будь-якими предметами;

– розміщувати на евакуаційних шляхах дзеркала та фальш-двері;

– встановлювати на шляхах евакуації будь-які конструктивні перешкоди: пороги, виступи, обертові, розсувні або підйомні двері;

– виконувати електрозварювальні та інші пожежонебезпечні роботи у присутності людей у будівлі;

– використовувати свічки, гасові лампи та ліхтарі для освітлення;

– застосовувати відкритий вогонь для відігріву труб систем опалення, водопостачання чи каналізації (допускається використання гарячої води, пари або гарячого піску);

– залишати використані обтиральні матеріали на робочому місці, у шафах або в кишенях робочого одягу;

– залишати без нагляду підключені до мережі електроприлади.

Сумлінне дотримання всіх перелічених вимог суттєво знижує ймовірність виникнення пожежі. Навіть якщо пожежа все ж таки сталася, виконання встановлених правил дозволяє організувати евакуацію більш оперативно та значно зменшити масштаби можливих наслідків.

ВИСНОВКИ

У результаті виконання магістерської роботи було проведено комплексне дослідження сучасних підходів до побудови архітектур вебклієнтів, визначено основні проблеми існуючих моделей фронтенд-розробки та сформовано методичні засади для створення адаптивної архітектурної методології ARX. Опрацьовано сучасні тенденції у сфері фронтенд-інженерії, критерії вибору архітектурних рішень, а також підходи до забезпечення продуктивності, підтримуваності, масштабованості та покращення досвіду розробника. Сформовано систему метрик для оцінювання ефективності архітектур вебзастосунків, що дало можливість виконати порівняльний аналіз сучасних фронтенд-моделей, зокрема SPA, та визначити їхні переваги й обмеження у контексті продуктивності, SEO, UX і організації процесу розробки.

На основі отриманих результатів було розроблено архітектурну методологію ARX, яка поєднує принципи адаптивного та гібридного рендерингу, модульності й автоматизації процесів розробки. Реалізовано базовий прототип системи з використанням React/Next.js та підтримкою серверних компонентів, а також досліджено можливості інтеграції ARX із AI-системами для автоматичної генерації scaffold-структур, тестування компонентів і адаптивного управління рендерингом. Експериментальне дослідження за допомогою Lighthouse, RUM та Puppeteer підтвердило ефективність запропонованої архітектури за показниками продуктивності, maintainability, scalability та developer experience. Отримані результати продемонстрували, що ARX покращує організацію коду, спрощує розподіл відповідальності між командами та створює передумови для ефективного масштабування сучасних вебзастосунків.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Amazon Web Services. (2024). AWS CloudFront Developer Guide. Retrieved from https://docs.aws.amazon.com/cloudfront/

2. Banks, A., & Porcello, E. (2020). Learning React. O'Reilly Media.

3. Biedenkapp, P. (2019). React Explained: Your Step-by-Step Guide to React.js. Independently Published.

4. Bodnarchuk, I., Skorenkyy, Y., Kramar, T., Duda, O., & Nykytyuk, V. (2022). Use of Analytical Hierarchy Process in Scenarios Design for a Digital Museum with XR components. The 2nd International Workshop on Information Technologies: Theoretical and Applied Problems (ITTAP-2022), Ternopil, Ukraine, November 22-24, 2022. Vol-3309, pp. 414-425. ISSN 1613-0073.

5. Chrome Team. (2023). Web Rendering Performance Best Practices. Google Chrome Labs.

6. Cloudflare Inc. (2024). Cloudflare Workers Documentation. Retrieved from https://developers.cloudflare.com/workers/

7. Cypress.io. (2024). Cypress Documentation. Retrieved from https://docs.cypress.io/

8. Datadog Inc. (2024). Real User Monitoring Guide. Retrieved from https://docs.datadoghq.com/real_user_monitoring/

9. Dediv, L., Dozorska, O., Kukuruza, V., Nykytyuk, V., & Kovalyk, S. (2023). Computer Simulation Modeling of Voice Signals in the Matlab Environment for the Task of Computerized Diagnostic Systems Testing. The 1st International Workshop on Computer Information Technologies in Industry 4.0 (CITI-2023), Ternopil, Ukraine, June 14-16, 2023. Vol-3468, pp. 257-262. ISSN 1613-0073.

10. Dozorskyi, V., Dediv, I., Sverstiuk, S., Nykytyuk, V., & Karnaukhov, A. (2023). The Method of Commands Identification to Voice Control of the Electric Wheelchair. The 1st International Workshop on Computer Information

Technologies in Industry 4.0 (CITI-2023), Ternopil, Ukraine, June 14-16, 2023. Vol-3468, pp. 233-240. ISSN 1613-0073.

11. Duda, O., Kunanets, N., Martsenko, S., Nykytyuk, V., & Pasichnyk, V. (2021). COVID-19 data collections and analytical processing. 2021 IEEE 16th International Conference on Computer Sciences and Information Technologies (CSIT), Vol. 2, Lviv, Ukraine, pp. 252-257. DOI: 10.1109/CSIT52700.2021.9648839.

12. Duda, O., Kunanets, N., Martsenko, S., Nykytyuk, V., & Pasichnyk, V. (2021). Information technology platform for the selection and analytical processing of information on COVID-19. 2021 IEEE 16th International Conference on Computer Sciences and Information Technologies (CSIT), Vol. 2, Lviv, Ukraine, pp. 231-238. DOI: 10.1109/CSIT52700.2021.9648839.

13. ECMA International. (2024). ECMAScript Language Specification. Retrieved from https://tc39.es/ecma262/

14. Express.js Team. (2024). Express Documentation. Retrieved from https://expressjs.com/

15. Facebook Open Source. (2024). React Documentation. Retrieved from https://react.dev

16. Figma Inc. (2024). Design Systems Best Practices. Retrieved from https://help.figma.com/

17. Fowler, M. (2019). Patterns of Enterprise Application Architecture. Addison-Wesley.

18. Freeman, E., & Robson, E. (2021). Head First Design Patterns. O'Reilly Media.

19. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (2019). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

20. GitHub Inc. (2024). GitHub Actions Documentation. Retrieved from https://docs.github.com/actions

21. GitLab Inc. (2024). GitLab CI/CD Documentation. Retrieved from https://docs.gitlab.com/ee/ci/

22. Google Developers. (2024). Chrome DevTools Documentation. Retrieved from https://developer.chrome.com/docs/devtools/

23. Google Developers. (2024). Core Web Vitals Guide. Retrieved from https://web.dev/vitals/

24. Google Developers. (2024). Lighthouse Documentation. Retrieved from https://developer.chrome.com/docs/lighthouse

25. Grinberg, M. (2021). Flask Web Development: Developing Web Applications with Python. O'Reilly Media.

26. Hanselman, S. (2019). Modern Web Development with JavaScript and TypeScript. Microsoft Press.

27. Jest Team. (2024). Jest Documentation. Retrieved from https://jestjs.io/docs/getting-started

28. Kent, C. D. (2020). Testing JavaScript Applications. Epic Web Publishing.

29. Koroliuk, R., Nykytyuk, V., Tymoshchuk, V., Soyka, V., & Tymoshchuk, D. (2024). Automated monitoring of bee colony movement in the hive during winter season. BAIT'2024: The 1st International Workshop on Bioinformatics and Applied Information Technologies, Zboriv, Ukraine, October 02-04, 2024. CEUR Workshop Proceedings, 3842, pp. 147-156. ISSN: 1613-0073.

30. Kryazhych, O., Itskovych, V., Iushchenko, K., Hrytsyshyna, V., Bruvier, D., Nykytyuk, V., & Bodnarchuk, I. (2023). The use of abstract moore automaton to control the sensors of a service-oriented alarm and emergency notification network. Scientific Journal of TNTU (Tern.), vol. 109, no. 1, pp. 111-120. ISSN 2522-4433.

31. Martin, R. C. (2017). Clean Code. Prentice Hall.

32. Martin, R. C. (2018). Clean Architecture. Prentice Hall.

33. Microsoft. (2024). Visual Studio Code Documentation. Retrieved from https://code.visualstudio.com/docs

34. MongoDB Inc. (2024). MongoDB Documentation. Retrieved from https://www.mongodb.com/docs/

35. Mozilla Foundation. (2024). MDN Web Docs: CSS Reference. Retrieved from https://developer.mozilla.org

36. Mozilla Foundation. (2024). MDN Web Docs: JavaScript Guide. Retrieved from https://developer.mozilla.org

37. NestJS Team. (2024). NestJS Documentation. Retrieved from https://docs.nestjs.com/

38. New Relic Inc. (2024). Browser Monitoring Documentation. Retrieved from https://docs.newrelic.com

39. Newman, S. (2021). Building Microservices. O'Reilly Media.

40. Node.js Foundation. (2024). Node.js Documentation. Retrieved from https://nodejs.org/docs/latest/api/

41. Nx DevTools. (2024). Nx Monorepo Documentation. Retrieved from https://nx.dev

42. Nykytyuk, V., Dozorskyi, V., Dozorska, O., Karnaukhov, A., & Matiichuk, L. (2022). The Method of User Identification by Speech Signal. The 2nd International Workshop on Information Technologies: Theoretical and Applied Problems (ITTAP-2022), Ternopil, Ukraine, November 22-24, 2022. Vol-3309, pp. 225-232. ISSN 1613-0073.

43. Nykytyuk, V., Dozorskyy, V., Kunanets, N., Pasichnyk, V., Matsiuk, O., & Bodnarchuk, I. (2021). Electrical Probe-Signal Processing and Criterion for the Determination of Time Parameters of the Teeth Filling Material Polymerization Process in Dentistry. 4th IDDM 2021, Valencia, Spain, pp. 54-63.

44. OpenAI. (2024). AI for Software Engineering Research Notes. Retrieved from https://openai.com

45. Osmani, A. (2019). JavaScript Design Patterns. Addy Osmani Publications.

46. Parcel Team. (2024). Parcel Documentation. Retrieved from https://parceljs.org/docs/

47. Playwright Team. (2024). Playwright Documentation. Retrieved from https://playwright.dev/docs/intro

48. PostgreSQL Global Development Group. (2024). PostgreSQL Documentation. Retrieved from https://www.postgresql.org/docs/

49. Puppeteer Team. (2024). Puppeteer Documentation. Retrieved from https://pptr.dev

50. Rauschmayer, A. (2018). Speaking JavaScript: An In-Depth Guide for Programmers. O'Reilly Media.

51. Redux Team. (2024). Redux Toolkit Documentation. Retrieved from https://redux-toolkit.js.org/

52. Richardson, C. (2018). Micro Frontends in Action. Manning Publications.

53. Rollup Team. (2024). Rollup Documentation. Retrieved from https://rollupjs.org/

54. Smashing Magazine. (2023). Modern Frontend Architecture Patterns. Retrieved from https://www.smashingmagazine.com/

55. Snyder, J. (2018). High Performance Browser Networking. O'Reilly Media.

56. SonarSource. (2024). SonarQube Documentation. Retrieved from https://docs.sonarsource.com

57. Storybook Team. (2024). Storybook Documentation. Retrieved from https://storybook.js.org/docs

58. Sverstiuk, A., Matiichuk, L., Polyvana, U., Stanko, A., & Nykytyuk, V. (2024). Analytical analysis of approaches to assessing the quality of life in smart cities. BAIT'2024: The 1st International Workshop on Bioinformatics and Applied Information Technologies, Zboriv, Ukraine, October 02-04, 2024. CEUR Workshop Proceedings, 3842, pp. 75-91. ISSN: 1613-0073.

59. TanStack. (2024). TanStack Query Documentation. Retrieved from https://tanstack.com/query/latest

60. Turborepo Team. (2024). Turborepo Documentation. Retrieved from https://turbo.build/repo/docs

61. TypeScript Team. (2024). TypeScript Documentation. Retrieved from https://www.typescriptlang.org/docs/

62. Vercel Inc. (2024). Edge Functions Documentation. Retrieved from https://vercel.com/docs/functions/edge-functions

63. Vercel. (2024). Next.js Documentation. Retrieved from https://nextjs.org/docs

64. Vernon, V. (2016). Domain-Driven Design Distilled. Addison-Wesley.

65. Vite Team. (2024). Vite Documentation. Retrieved from https://vitejs.dev/guide/

66. W3C. (2024). CSS Specifications. Retrieved from https://www.w3.org/Style/CSS/

67. W3C. (2024). HTML Standard. Retrieved from https://html.spec.whatwg.org

68. Web Performance Working Group. (2018). Web Performance: Optimizing for Speed and User Experience. W3C.

69. Webpack Team. (2024). Webpack Documentation. Retrieved from https://webpack.js.org/concepts/

70. Yasniy, O., Didych, I., Tymoshchuk, D., Pasternak, I., Nykytyuk, V., Shymchuk, H., & Radyk, D. (2026). Fatigue crack growth prediction of automotive steels using ensemble-based machine learning methods. Procedia Structural Integrity, VIII International Conference In-service Damage of Materials: Diagnostics and Prediction, Vol. 81, pp. 116-122.

71. Zakas, N. C. (2020). JavaScript: The Good Parts. O'Reilly Media.

ДОДАТКИ

Додаток А - фрагмент коду з компонент ARX News Header

```
"use client";

import React, { useMemo, useState, useEffect } from "react";

/**
 * ARX: Context layer (runtime signals)
 */
type RenderMode = "SSR" | "CSR" | "HYBRID";

interface ARXContext {
  isLoggedIn: boolean;
  device: "mobile" | "desktop";
  network: "slow" | "fast";
  region?: string;
  enablePersonalization?: boolean;
}

/**
 * Domain layer (News)
 */
interface Category {
  id: string;
  label: string;
}

interface Notification {
  id: string;
  title: string;
  type: "breaking" | "normal";
}

interface User {
  name: string;
  avatar?: string;
  bookmarksCount: number;
}

/**
 * ARX Policy Layer (adaptive rendering decision)
 */
function resolveRenderMode(ctx: ARXContext): RenderMode {
  if (ctx.network === "slow") return "SSR";
  if (ctx.device === "mobile" && !ctx.enablePersonalization) return "HYBRID";
  return "CSR";
}

/**
 * Simulated API layer (would be SSR/edge in real ARX)
 */
async function fetchBreakingNews(): Promise<Notification[]> {
  return [
    { id: "1", title: "Market crashes after AI regulation update", type:
"breaking" },
    { id: "2", title: "Global tech index rises", type: "normal" }
```

```
  ];
}

async function fetchCategories(): Promise<Category[]> {
  return [
    { id: "world", label: "World" },
    { id: "tech", label: "Tech" },
    { id: "business", label: "Business" },
    { id: "sports", label: "Sports" }
  ];
}

/**
 * UI layer (pure presentation)
 */
function HeaderView({
  categories,
  notifications,
  user,
  searchQuery,
  setSearchQuery,
}: {
  categories: Category[];
  notifications: Notification[];
  user?: User;
  searchQuery: string;
  setSearchQuery: (v: string) => void;
}) {
  return (
    <header style={{ display: "flex", flexDirection: "column", gap: 12 }}>
      {/* Top bar */}
      <div style={{ display: "flex", justifyContent: "space-between" }}>
        <div style={{ fontWeight: 700 }}>📰 ARX News</div>

        <input
          placeholder="Search news..."
          value={searchQuery}
          onChange={(e) => setSearchQuery(e.target.value)}
          style={{ padding: 6, width: "40%" }}
        />

        <div style={{ display: "flex", gap: 10 }}>
          {user ? (
            <>
              <span>👤 {user.name}</span>
              <span>🔖 {user.bookmarksCount}</span>
            </>
          ) : (
            <button>Login</button>
          )}
        </div>
      </div>

      {/* Breaking news ticker */}
      <div style={{ background: "#111", color: "#fff", padding: 6 }}>
        <strong>Breaking:</strong>{" "}
        {notifications.find((n) => n.type === "breaking")?.title ??
```

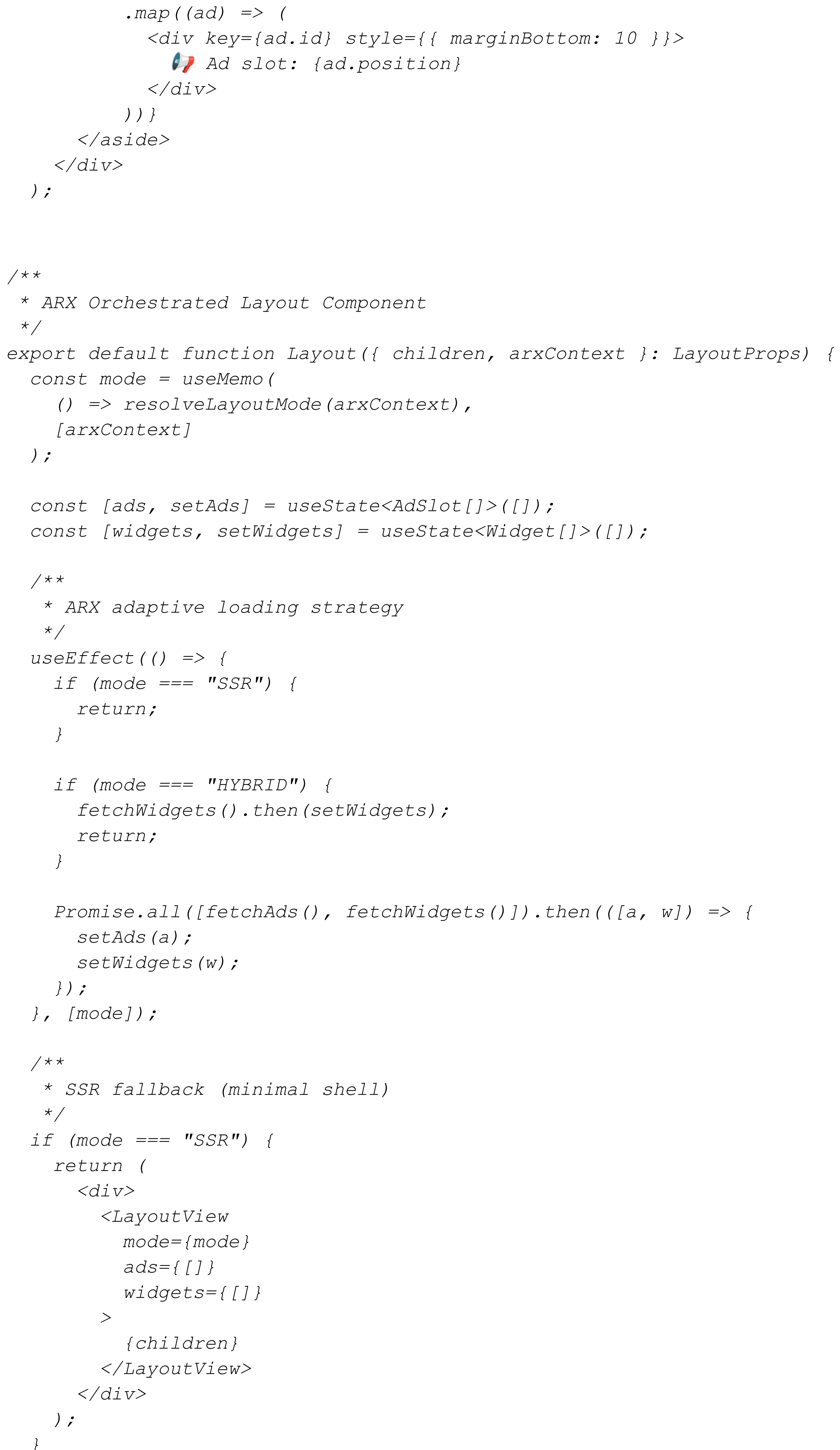
```
          "No breaking news"}
      </div>

      {/* Categories navigation */}
      <nav style={{ display: "flex", gap: 12 }}>
        {categories.map((c) => (
          <button key={c.id}>{c.label}</button>
        ))}
      </nav>
    </header>
  );
}

/**
 * ARX Orchestrated Component (core logic layer)
 */
export default function Header({
  arxContext,
  user,
}: {
  arxContext: ARXContext;
  user?: User;
}) {
  const mode = useMemo(
    () => resolveRenderMode(arxContext),
    [arxContext]
  );

  const [categories, setCategories] = useState<Category[]>([]);
  const [notifications, setNotifications] = useState<Notification[]>([]);
  const [searchQuery, setSearchQuery] = useState("");

  /**
   * ARX: adaptive data loading strategy
   */
  useEffect(() => {
    if (mode === "SSR") {
      // minimal client hydration
      return;
    }

    if (mode === "HYBRID") {
      fetchCategories().then(setCategories);
      return;
    }

    // CSR mode (full interactivity + personalization)
    Promise.all([fetchCategories(), fetchBreakingNews()]).then(
      ([cats, notes]) => {
        setCategories(cats);
        setNotifications(notes);
      }
    );
  }, [mode]);

  /**
   * ARX: fallback rendering strategy
```

```
   */
  if (mode === "SSR") {
    return (
      <div>
        <HeaderView
          categories={[]}
          notifications={[]}
          user={user}
          searchQuery=""
          setSearchQuery={() => {}}
        />
      </div>
    );
  }

  return (
    <div data-arx-mode={mode}>
      <HeaderView
        categories={categories}
        notifications={notifications}
        user={user}
        searchQuery={searchQuery}
        setSearchQuery={setSearchQuery}
      />
    </div>
  );
}
```

Додаток Б - Фрагмент коду компоненту ARX News Layout

```
"use client";

import React, { useMemo, useEffect, useState } from "react";

/**
 * ARX Core Types
 */
type RenderMode = "SSR" | "CSR" | "HYBRID";

interface ARXContext {
  device: "mobile" | "desktop";
  network: "slow" | "fast";
  isLoggedIn: boolean;
  region?: string;
  contentDensity?: "low" | "medium" | "high";
}

/**
 * Layout domain types
 */
interface LayoutProps {
  children: React.ReactNode;
  arxContext: ARXContext;
}

interface AdSlot {
  id: string;
  position: "top" | "side" | "footer";
  active: boolean;
}

interface Widget {
  id: string;
  type: "weather" | "stocks" | "trending";
  data: any;
}

/**
 * ARX Policy Layer (Layout rendering strategy)
 */
function resolveLayoutMode(ctx: ARXContext): RenderMode {
  if (ctx.network === "slow") return "SSR";
  if (ctx.device === "mobile") return "HYBRID";
  if (ctx.contentDensity === "high") return "HYBRID";
  return "CSR";
}

/**
 * Simulated infrastructure layer (edge/SSR data)
 */
async function fetchAds(): Promise<AdSlot[]> {
  return [
    { id: "ad-top", position: "top", active: true },
    { id: "ad-side", position: "side", active: true }
```

```
  ];
}

async function fetchWidgets(): Promise<Widget[]> {
  return [
    { id: "weather", type: "weather", data: { temp: 18 } },
    { id: "stocks", type: "stocks", data: { sp500: "+1.2%" } },
    { id: "trending", type: "trending", data: ["AI", "React", "ARX"] }
  ];
}

/**
 * UI Layer (pure layout rendering)
 */
function LayoutView({
  children,
  ads,
  widgets,
  mode,
}: {
  children: React.ReactNode;
  ads: AdSlot[];
  widgets: Widget[];
  mode: RenderMode;
}) {
  return (
    <div style={{ display: "grid", gridTemplateColumns: "250px 1fr 250px" }}>
      {/* Left sidebar widgets */}
      <aside>
        <h4>Widgets</h4>
        {widgets.map((w) => (
          <div key={w.id} style={{ marginBottom: 10 }}>
            {w.type === "weather" && <div> {w.data.temp}°C</div>}
            {w.type === "stocks" && <div> {w.data.sp500}</div>}
            {w.type === "trending" && (
              <div> {w.data.join(", ")}</div>
            )}
          </div>
        ))}
      </aside>

      {/* Main content */}
      <main>
        {mode === "HYBRID" && (
          <div style={{ background: "#f5f5f5", padding: 8 }}>
             Hybrid rendering active
          </div>
        )}

        {children}
      </main>

      {/* Right sidebar ads */}
      <aside>
        <h4>Ads</h4>
        {ads
          .filter((a) => a.active)
```



```
          .map((ad) => (
            <div key={ad.id} style={{ marginBottom: 10 }}>
              📢 Ad slot: {ad.position}
            </div>
          ))}
        </aside>
      </div>
    );


/**
 * ARX Orchestrated Layout Component
 */
export default function Layout({ children, arxContext }: LayoutProps) {
  const mode = useMemo(
    () => resolveLayoutMode(arxContext),
    [arxContext]
  );

  const [ads, setAds] = useState<AdSlot[]>([]);
  const [widgets, setWidgets] = useState<Widget[]>([]);

  /**
   * ARX adaptive loading strategy
   */
  useEffect(() => {
    if (mode === "SSR") {
      return;
    }

    if (mode === "HYBRID") {
      fetchWidgets().then(setWidgets);
      return;
    }

    Promise.all([fetchAds(), fetchWidgets()]).then(([a, w]) => {
      setAds(a);
      setWidgets(w);
    });
  }, [mode]);

  /**
   * SSR fallback (minimal shell)
   */
  if (mode === "SSR") {
    return (
      <div>
        <LayoutView
          mode={mode}
          ads={[]}
          widgets={[]}
        >
          {children}
        </LayoutView>
      </div>
    );
  }
```

```
  return (
    <div data-arx-layout-mode={mode}>
      <LayoutView mode={mode} ads={ads} widgets={widgets}>
        {children}
      </LayoutView>
    </div>
  );
}
```

Міністерство освіти і науки України
Тернопільський національний технічний університет
імені Івана Пулюя
Маріборський університет (Словенія)
Технічний університет в Кошице (Словаччина)
Каунаський технологічний університет (Литва)
Львівський національний університет
імені Івана Франка
Гірничо-металургійна академія ім. Станіслава Сташиця (Польща)
Луцький національний технічний університет
Чернівецький національний університет
імені Юрія Федьковича
Вроцлавський економічний університет (Польща)
Університет технологій та економіки
імені Хелени Ходковської (Польща)
Донбаська державна машинобудівна академія

Студентське наукове товариство

ІХ МІЖНАРОДНА

студентська науково - технічна конференція

"ПРИРОДНИЧІ ТА ГУМАНІТАРНІ НАУКИ. АКТУАЛЬНІ ПИТАННЯ"

24-25 квітня 2026 р.

(збірник тез конференції)

Тернопіль 2026

УДК 004.42

Никитюк В., канд.техн.наук; Старицький О.
Тернопільський національний технічний університет імені Івана Пулюя

ОПТИМІЗАЦІЯ АДАПТИВНО-ГІБРИДНОЇ АРХІТЕКТУРИ ARX ДЛЯ ПІДВИЩЕННЯ МАСШТАБОВАНОСТІ ТА ПРОДУКТИВНОСТІ ВЕБКЛІЄНТІВ

Nykytiuk V., Starytskyi O.
Ternopil Ivan Puluj National Technical University

OPTIMIZATION OF THE ADAPTIVE-HYBRID ARX ARCHITECTURE FOR IMPROVING THE SCALABILITY AND PERFORMANCE OF WEB CLIENTS

Оптимізація архітектури фронтенд-застосунків відіграє важливу роль у забезпеченні їхньої масштабованості та продуктивності. В умовах зростання складності вебклієнтів і необхідності роботи в реальному часі традиційні підходи не завжди забезпечують достатню гнучкість та ефективність.

Метою роботи є дослідження адаптивно-гібридної архітектури ARX та аналіз можливостей її оптимізації для підвищення продуктивності й масштабованості вебклієнтів. Особливу увагу приділено управлінню станом, розподілу логіки між клієнтом і сервером та адаптації архітектури до змін навантаження.

Для досягнення мети проаналізовано існуючі архітектурні підходи у фронтенд-розробці та визначено їхні переваги й недоліки. Результати порівняння монолітних, серверних і мікрофронтенд-рішень наведено в таблиці 1, що дозволило виявити обмеження традиційних підходів і обґрунтувати доцільність використання гібридної моделі.

Таблиця 1. Порівняння архітектур фронтенд-застосунків

Напрямок оптимізації	Опис	Вплив на систему
Lazy loading	Завантаження по мірі потреби	↓ час завантаження
Code splitting	Розділення bundle	↓ навантаження
SSR/CSR hybrid	Комбінування рендерингу	↑ SEO + швидкість
Caching	Кешування даних	↑ продуктивність
State optimization1	Оптимізація стану	↑ масштабованість

Провівши порівняння та ознайомившись із функціоналом кожного з рендерів, обрано SSR, оскільки цей підхід дозволяє ефективно вирішувати задачі SEO та працювати з динамічним контентом. Він забезпечує генерацію готового HTML на

IX Міжнародна студентська науково - технічна конференція
"ПРИРОДНИЧІ ТА ГУМАНІТАРНІ НАУКИ. АКТУАЛЬНІ ПИТАННЯ"

УДК 004.42
Никитюк В., канд.техн.наук; Старицький О.
Тернопільський національний технічний університет імені Івана Пулюя

ОПТИМІЗАЦІЯ ІНФОРМАЦІЙНОЇ АРХІТЕКТУРИ ДЛЯ ВЕБЗАСТОСУНКІВ ІЗ ДИНАМІЧНИМИ ДАНИМИ ТА AI-КОМПОНЕНТАМИ

Nykytiuk V., Starytskyi O.
Ternopil Ivan Puluj National Technical University

DEVELOPMENT OF INFORMATION ARCHITECTURE FOR WEB APPLICATIONS WITH DYNAMIC DATA AND AI COMPONENTS

Оптимізація інформаційної архітектури вебзастосунків відіграє важливу роль у забезпеченні їхньої ефективності, масштабованості та зручності використання. В умовах зростання обсягів динамічних даних та інтеграції AI-компонентів традиційні підходи до організації інформації вже не забезпечують необхідної гнучкості та керованості системи.

Метою роботи є дослідження підходів до побудови інформаційної архітектури вебзастосунків із динамічними даними та AI-компонентами, а також аналіз можливостей її оптимізації для підвищення ефективності взаємодії користувача з системою. Особливу увагу приділено структуризації даних, організації потоків інформації та інтеграції AI-рішень у загальну архітектуру.

Для досягнення поставленої мети було проаналізовано існуючі підходи до побудови інформаційної архітектури, визначено їхні переваги та недоліки в контексті роботи з динамічними даними та AI. Результати аналізу дозволили виявити обмеження традиційних моделей і обґрунтувати необхідність використання більш гнучких та адаптивних підходів до організації інформаційних структур.

Таблиця 1. Модель інформаційної архітектури вебзастосунку

Компонент інформаційної архітектури	Призначення	Роль у системі
Data sources (API, DB, AI services)	Джерела структурованих і неструктурованих даних	Забезпечують основу для формування контенту
Data aggregation layer	Агрегація даних з різних джерел	Формує єдину модель даних для UI
Business logic layer	Обробка правил, сценаріїв та AI-інтеграцій	Визначає поведінку системи
Presentation layer (UI)	Відображення даних користувачу	Забезпечує взаємодію та UX
AI/ML integration layer	Інтеграція моделей, embeddings, inference	Додає інтелектуальну обробку та персоналізацію

Провівши порівняння існуючих підходів до побудови інформаційної архітектури, зокрема традиційних ієрархічних моделей, компонентно-орієнтованих

Семенів М. **ДОСЛІДЖЕННЯ ТА РОЗРОБКА МОДЕЛЕЙ ГЛИБИННОГО НАВЧАННЯ ДЛЯ КЛАСИФІКАЦІЇ ХВОРОБ РОСЛИН ЗА ЗОБРАЖЕННЯМИ**	**229**
Смик А. **РОЗРОБКА ВЕБ-ПЛАТФОРМИ ДЛЯ УПРАВЛІННЯ ЗАДАЧАМИ ТА АНАЛІЗУ ПРОДУКТИВНОСТІ**	**231**
Никитюк В., Старицький О. **ОПТИМІЗАЦІЯ ІНФОРМАЦІЙНОЇ АРХІТЕКТУРИ ДЛЯ ВЕБЗАСТОСУНКІВ ІЗ ДИНАМІЧНИМИ ДАНИМИ ТА АІ-КОМПОНЕНТАМИ**	**233**
Никитюк В., Старицький О. **ОПТИМІЗАЦІЯ АДАПТИВНО-ГІБРИДНОЇ АРХІТЕКТУРИ ARX ДЛЯ ПІДВИЩЕННЯ МАСШТАБОВАНОСТІ ТА**	**235**
Стремецький П. **ІНСТРУМЕНТИ ТА ПРАКТИКИ ДЛЯ АНАЛІЗУ ЯКОСТІ ВИХІДНОГО КОДУ JAVA- ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ**	**237**
Сумко В. **МЕТОДИ ОЦІНЮВАННЯ РИТМУ ЕЛЕКТРОКАРДІОСИГНАЛІВ**	**238**
Тиховліс Р. **МОДЕЛІ ДЛЯ ПРЕДСТАВЛЕННЯ ЕКОНОМІЧНИХ ДАНИХ**	**240**
Тиховліс Р. **ПІДХОДИ ДО МОДЕЛЮВАННЯ ЕКОНОМІЧНИХ ДАНИХ**	**241**
Хоренко В. **РОЗРОБКА ІНТЕЛЕКТУАЛЬНОГО АІ-АСИСТЕНТА З КОНСУЛЬТУВАННЯ МОВИ ПРОГРАМУВАННЯ PYTHON НА ОСНОВІ QDRANT ТА GEMMA3**	**242**
Целінь А. **СТВОРЕННЯ ІНТЕЛЕКТУАЛЬНОЇ ПРОГРАМНОЇ СИСТЕМИ ДЛЯ АВТОМАТИЧНОГО ВИЗНАЧЕННЯ ПОРУШЕНЬ КОНЦЕНТРАЦІЇ УВАГИ**	**244**
Чайківський С. **МЕТОДИ РОЗПІЗНАВАННЯ МУЗИЧНИХ АКОРДІВ ЗАСОБАМИ МАШИННОГО НАВЧАННЯ**	**246**
Чигрин М. **АРХІТЕКТУРНІ ПІДХОДИ ДО ПОБУДОВИ ВЕБ-СИСТЕМИ УПРАВЛІННЯ НАВЧАЛЬНИМ КОНТЕНТОМ**	**248**
Чорнописький Б. **ПРОГРАМНА РЕАЛІЗАЦІЯ ІНФОРМАЦІЙНОЇ СИСТЕМИ АНАЛІЗУ ТА ВИБОРУ ІТ-РІШЕНЬ**	**250**