Міністерство освіти і науки України
Криворізький національний університет
Кафедра моделювання та програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття ступеня вищої освіти бакалавра

зі спеціальності 121 − Інженерія програмного забезпечення

На тему: Розробка веб-застосунку для потокового відтворення відео

Засвідчую, що в цій кваліфікаційній роботі немає запозичень із праць інших авторів без відповідних посилань.

Студент групи ІПЗ-22-1

_____ / В. О. Ярошенко/

Керівник
кваліфікаційної роботи / О. Г. Рибальченко /
Завідувач кафедри / А. М. Стрюк /

Кривий Ріг
2026

Криворізький національний університет

Факультет: Інформаційних технологій

Кафедра Моделювання та програмного забезпечення

Ступінь вищої освіти: бакалавр

Спеціальність: 121 − Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Зав. кафедрою
к.п.н., доц. А.М. Стрюк
“____”_____20___р.

ЗАВДАННЯ

на кваліфікаційну роботу

студенту групи ІПЗ-22-1 Ярошенко Віталію Олеговичу

1. Тема: Розробка веб-застосунку для потокового відтворення відео затверджено наказом по КНУ № 284-с від «28» травня 2026 р.
2. Термін подання студентом закінченої роботи: «10» червня 2026 р.
3. Вихідні дані по роботі: Результати аналізу існуючих платформ відеострімінгу та хостингів (YouTube, Twitch, TikTok); вимоги до побудови інтерактивного інтерфейсу «кокпіта» та алгоритмів високоточної синхронізації подій у реальному часі (Event-to-UI sync); архітектурні рішення модульного моноліту застосунку та проєктування реляційної моделі бази даних; технічна документація використаних технологій та інструментів (TypeScript, NestJS, React, PostgreSQL, Prisma, FFmpeg, BullMQ, Redis, Socket.IO, Cloudflare R2, Vidstack).
4. Зміст пояснювальної записки (перелік питань, що їх треба розробити): Вступ. Розділ 1 – Аналіз сучасних технологій інтерактивного стрімінгу та обґрунтування актуальності розробки. Висвітлено ринковий контекст, проблему стагнації інтерфейсів та обмеження аналогів у вигляді YouTube,

Twitch і TikTok, а також сформовано актуальність, мету й завдання проєкту «Amplify». Розділ 2 – Проєктний аналіз та моделювання вебзастосунку. Описано розподіл ролей доступу за моделлю RBAC, побудову структурної схеми підсистем, UML-діаграму прецедентів та функціональне моделювання процесів за стандартом IDEF0. Представлено діаграму послідовностей для асинхронної обробки медіа та реляційну модель бази даних у третій нормальній формі. Розділ 3 – Програмна реалізація модулів проєкту. Обґрунтовано використання монорепозиторію на базі TypeScript, розробку сервера на NestJS 10 та взаємодію з PostgreSQL через Prisma 6. Розглянуто фонове транскодування у формат HLS за допомогою FFmpeg з чергами BullMQ, а також клієнтську частину на React 19 із реал-тайм синхронізацією через Socket.IO 4 та сховище Cloudflare R2. Розділ 4 – Дослідження використання розробленого застосунку. Наведено огляд макета інтерфейсу, механізмів дебаунсу, автентифікації Google OAuth 2.0 та сторінки каналу. Описано роботу сторінки перегляду відео з віджетами «кокпіта», інтерактивними коментарями та функціонування студії автора з часовою шкалою Drag-and-Drop. Висновки. Список використаних джерел.

5. Перелік ілюстративного матеріалу: структурна схема, діаграма варіантів використання (прецедентів), контекстна діаграма, діаграма послідовностей, схема бази даних, знімки екрану розробленого вебзастосунку.

Календарний план:

№	Найменування етапів кваліфікаційної роботи	Термін виконання етапів роботи
1	Огляд літературних джерел відповідно до теми кваліфікаційної роботи	31.03.2026 – 04.04.2026
2	Пошук та проведення аналізу існуючих у предметній області аналогів програмних продуктів	05.04.2026 – 08.04.2026
3	Формулювання актуальності та мети і визначення завдань роботи	09.04.2026 – 11.04.2026
4	Підготовка матеріалів першого розділу роботи	12.04.2026 – 16.04.2026
5	Розробка діаграм архітектури програмного продукту, проєктування бази даних	17.04.2026 – 24.04.2026
6	Підготовка матеріалів другого розділу роботи	25.04.2026 – 29.04.2026
7	Створення і наповнення бази даних, розробка back-end частини вебзастосунку	30.04.2026 – 09.05.2026
8	Вибір технологій реалізації проєкту, розробка front-end частини вебзастосунку	10.05.2026 – 18.05.2026
9	Підготовка матеріалів третього розділу роботи	19.05.2026 – 22.05.2026
10	Тестування розробленого програмного продукту, усунення виявлених дефектів	23.05.2026 – 26.05.2026
11	Підготовка матеріалів четвертого розділу роботи	27.05.2026 – 30.05.2026
12	Оформлення пояснювальної записки	31.05.2026 – 04.06.2026

Дата видачі завдання: «30» березня 2026р.

Студент: _____ / В. О. Ярошенко/

Керівник роботи: _____ / О. Г. Рибальченко /

РЕФЕРАТ

ВЕБ-ЗАСТОСУНОК, ПОТОКОВЕ ВІДТВОРЕННЯ ВІДЕО, ВІДЕОХОСТИНГ, REACT, NEST.JS, TYPESCRIPT

Пояснювальна записка: 55 с., 1 табл., 8 дод., 20 рис., 23 джерел.

Головна мета роботи полягає в розробці веб-застосунку «Amplify» для інтерактивного стрімінгу, який перетворює пасивний перегляд відео на динамічну роботу з контентом через концепцію «кокпіта». Його головна особливість полягає в тому, що інтерфейс плеєра не є статичним.

Об’єктом проектування є веб-застосунок з архітектурою модульного моноліту, що забезпечує функціонування інтерактивного інтерфейсу для персоналізованого досвіду взаємодії, де конфігурація робочого простору користувача динамічно адаптується до вхідного потоку метаданих у реальному часі.

Предметом проектування є методи та алгоритми синхронізації стану відеопотоку з динамічними компонентами інтерфейсу, а також архітектурні рішення для забезпечення низької затримки при передачі метаданих між серверною та клієнтською частинами.

У межах даної кваліфікаційної роботи було проведено глибокий аналіз принципів побудови сучасних вебресурсів для передачі медіаконтенту та методів організації користувацького досвіду. Дослідження було зосереджене на вивченні існуючих моделей взаємодії, що дозволило виявити характерні переваги та системні обмеження традиційних інтерфейсів. Зокрема, було проаналізовано ефективність статичних навігаційних елементів та їхню здатність задовольняти потреби користувача в умовах динамічної зміни контексту відеопотоку. Отримані результати підтвердили, що жорстка фіксація функціональних зон навколо відеоплеєра часто створює надлишкове когнітивне навантаження або, навпаки, не забезпечує достатнього інструментарію для повноцінної роботи з контентом у реальному часі.

ABSTRACT

WEB APPLICATION, VIDEO STREAMING, VIDEO HOSTING, REACT, NEST.JS, TYPESCRIPT

Explanatory note: 55 pp., 1 tables, 8 appendices, 20 figs., 23 sources.

The main objective of this project is to develop the “Amplify” web application for interactive streaming, which transforms passive video viewing into dynamic content interaction through the “cockpit” concept. Its key feature is that the player interface is not static.

The design object is a web application with a modular monolith architecture that enables an interactive interface for a personalized user experience, where the user’s workspace configuration dynamically adapts to the incoming metadata stream in real time.

The subject of the design is methods and algorithms for synchronizing the video stream’s state with dynamic interface components, as well as architectural solutions to ensure low latency when transferring metadata between the server and client sides.

Within the scope of this qualification work, an in-depth analysis of the principles of constructing modern web resources for media content delivery and methods of organising user experience was conducted. The research focused on studying existing interaction models, which allowed for the identification of characteristic advantages and systemic limitations of traditional interfaces. In particular, the effectiveness of static navigation elements and their ability to satisfy user needs in conditions of dynamic changes in the video stream context were analysed. The results obtained confirmed that the rigid fixation of functional zones around the video player often creates excessive cognitive load or, conversely, fails to provide sufficient tools for full-scale real-time interaction with the content.

ЗМІСТ

ВСТУП 8
1. АНАЛІЗ СУЧАСНИХ ТЕХНОЛОГІЙ ІНТЕРАКТИВНОГО СТРІМІНГУ ТА ОБҐРУНТУВАННЯ АКТУАЛЬНОСТІ РОЗРОБКИ 10
1.1. Аналіз професійної області проблеми 10
1.2. Аналіз існуючих аналогів 11
1.3. Формування актуальності і завдань роботи 15
2. ПРОЄКТУВАННЯ ВЕБЗАСТОСУНКУ 17
2.1. Визначення користувацьких ролей і рівнів доступу до системи 17
2.2. Створення структурної схеми та діаграми прецедентів 18
2.3. Розробка функціональної діаграми 25
2.4. Створення діаграми послідовностей 28
2.5. Побудова моделі «сутність-зв’язок» бази даних вебзастосунку 31
3. ПРОГРАМНА РЕАЛІЗАЦІЯ МОДУЛІВ ПРОЕКТУ 34
3.1. Вимоги до системи та базовий архітектурний підхід 34
3.2. Серверна інфраструктура та управління даними 36
3.3. Клієнтська архітектура та користувацький інтерфейс 37
3.4. Технології обробки мультимедіа та синхронізації в реальному часі 39
4. ВИКОРИСТАННЯ РОЗРОБЛЕНОГО ЗАСТОСУНКУ 41
4.1. Загальний огляд інтерфейсу та навігація 41
4.2. Реєстрація, авторизація та управління профілем 43
4.3. Перегляд відео та інтерактивні віджети 45
4.4. Завантаження та редагування відео 48
ВИСНОВОК 52
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 54

ВСТУП

Цифрова трансформація людського суспільства у XXI сторіччі перетворила взаємодію з інформаційними технологіями на невід'ємний аспект повсякденного життя. Від моменту появи перших систем телемовлення медіаконтент став основним каналом отримання знань, розваг та комунікації. Проте, якщо на початкових етапах розвитку технологій глядач залишався пасивним отримувачем сигналу, то сучасна епоха вимагає принципово нового рівня залученості – інтерактивності.

Еволюція вебтехнологій та поява глобальних відеоплатформ, таких як YouTube, ознаменували перехід до епохи двосторонньої комунікації. Користувачі отримали можливість не лише споживати контент, а й активно впливати на його життєвий цикл через механізми зворотного зв'язку: систему рейтингів, коментарів та соціальних реакцій. Проте, попри кількісне зростання інструментів взаємодії, вони здебільшого залишаються відокремленими від самого відеопотоку. Сучасний етап розвитку диктує потребу в глибшій, контекстній інтерактивності, де інтерфейс перестає бути лише статичною рамкою навколо плеєра та починає динамічно реагувати на зміст контенту в реальному часі.

Існуючі архітектурні підходи до побудови медіаресурсів часто обмежуються базовими графічними накладенннями або фіксованими текстовими зонами, які функціонують відірвано від внутрішньої хронології та семантики відео. Це створює просторове та когнітивне роз'єднання, змушуючи глядача розсіювати увагу між переглядом та пошуком релевантних інструментів в інших частинах екрана. Відсутність механізмів гнучкої адаптації робочого простору під конкретний момент часу суттєво обмежує потенціал використання відеоматеріалів у таких критично важливих сферах, як дистанційне навчання, технічний моніторинг чи інтерактивна комерція, де точність і своєчасність синхронізації даних є вирішальними.

Головна мета даної роботи полягає у розробці інноваційного вебзастосунку «Amplify», що реалізує концепцію динамічного інтерактивного «кокпіта» для стрімінгу медіаконтенту. Основна ідея проєкту фокусується на трансформації традиційного статичного плеєра в інтелектуальне середовище, яке здатне в реальному часі адаптувати свій інтерфейс, розгортаючи контекстно залежні віджети відповідно до подій у відеопотоці. Це дозволяє забезпечити користувачу персоналізований досвід взаємодії, де кожен елемент керування є функціонально виправданим та синхронізованим із поточним змістом контенту.

Для досягнення поставленої мети передбачено розв’язання низки стратегічних завдань. Першочергово це стосується проектування гнучкої модульної архітектури, здатної обробляти потоки метаданих із мінімальною затримкою, а також розробки алгоритмів синхронізації часових міток відео з активацією клієнтських компонентів. Окрім цього, у межах роботи створюється динамічна система віджетів на базі сучасних вебтехнологій. Особлива увага приділяється забезпеченню високої продуктивності та масштабованості системи для її подальшої інтеграції в різні прикладні сфери. Практична цінність результатів дослідження полягає у створенні універсального архітектурного патерну для побудови медіаресурсів нового покоління, які оптимізують когнітивне навантаження на користувача та трансформують споживання інформації у повноцінний цифровий робочий простір.

1. АНАЛІЗ СУЧАСНИХ ТЕХНОЛОГІЙ ІНТЕРАКТИВНОГО СТРІМІНГУ ТА ОБҐРУНТУВАННЯ АКТУАЛЬНОСТІ РОЗРОБКИ

1.1. Аналіз професійної області проблеми

Дослідження К. Фукса [1] демонструє, що на сучасному етапі розвитку цифрової індустрії ринок відеохостингів та стрімінгових сервісів демонструє виражену технологічну стагнацію, зумовлену монополізацією галузі декількома корпоративними гігантами. Глобальний аналіз професійного середовища, який проводить Т. Ву [2], свідчить про те, що вектор розвитку провідних платформ (YouTube, Twitch, TikTok) остаточно змістився від впровадження нових функціональних інструментів до агресивної оптимізації алгоритмів утримання уваги та максимізації рекламних доходів. В гонці за показниками «Retention Rate» та «Watch Time» інновації в архітектурі користувацького інтерфейсу були принесені в жертву алгоритмічній рекурентності. Як наслідок, відеоплеєр перетворився зі складного інструменту роботи з інформацією на спрощений засіб доставки контенту, головна мета якого – безперервне відтворення реклами та автоматично підібраних рекомендацій.

Системною проблемою сучасних платформ є їхній технологічний консерватизм. Величезні витрати на підтримку інфраструктури зберігання даних та CDN-мереж змушують власників сервісів дотримуватися максимально «безпечних» та перевірених моделей інтерфейсу. Будь-які спроби радикальної зміни структури взаємодії вважаються фінансово ризикованими, що призводить до повної відсутності виразних рис у аналогів. Фактично, користувач спостерігає «дизайн-вакуум»: незалежно від обраного ресурсу, він отримує ідентичний набір інструментів, який не змінювався по суті вже понад десять років. Це створює умови, за яких розвиток платформ відбувається виключно в площині «бек-енд» оптимізації рекламних кабінетів, тоді як «фронт-енд» можливості для творців та глядачів залишаються заблокованими в рамках застарілих стандартів.

Додатковим викликом є деградація функціональної складової медіасервісів на користь розважальної. Як зазначають М. Флайєлл та співавтори [3], ставка на пасивне споживання («scrolling» та «binge-watching») витіснила з пріоритетів розробки інструментарій для професійного, освітнього або аналітичного використання відеоконтенту. Фундаментальні дослідження когнітивного залучення, зокрема роботи М. Чі та Р. Вайлі [4], доводять, що пасивне спостереження за медіа суттєво обмежує глибину аналізу, тоді як інтерактивне та конструктивне середовище є критично необхідним для повноцінної роботи з інформацією. Сьогоднішні платформи не зацікавлені в тому, щоб глядач «працював» із відео, оскільки активна взаємодія та аналіз даних можуть перервати потік автоматичного споживання. Така монополія на увагу та відсутність інструментальної глибини в існуючих аналогах створюють значний розрив між потребами сучасного інтелектуального користувача та реальними можливостями сервісів. Глядач опиняється в ситуації, де він не має жодного впливу на структуру медіасередовища, а його роль зведена до об’єкта для показу реклами.

1.2. Аналіз існуючих аналогів

Для глибокого розуміння ринкового контексту та технологічного розриву, який покликаний заповнити проєкт «Amplify», необхідно детально розглянути ключових гравців сучасної медіаіндустрії. Кожен із обраних аналогів представляє певний етап еволюції відеосервісів, проте всі вони демонструють спільну рису – обмеженість у питаннях гнучкої трансформації користувацького інтерфейсу під конкретні потреби глядача.

Першим і найбільш репрезентативним об'єктом аналізу є YouTube. Як глобальний лідер галузі, ця платформа встановила стандарти дистрибуції відео, проте за останні десять років її інтерфейсна модель практично не зазнала фундаментальних змін. Головна проблема YouTube полягає у жорсткій фіксації відеоплеєра всередині статичної сторінки, де оточуючі блоки (опис, коментарі, рекомендації) існують автономно від самого відеопотоку. Будь-які спроби

впровадження інтерактивності, як-от «підказки» чи «кінцеві заставки», мають примітивний характер і часто сприймаються користувачами як візуальний шум, оскільки вони просто нашаровуються поверх кадру, перекриваючи важливий контент. Платформа повністю орієнтована на пасивне споживання, де роль глядача зведена до кліків по алгоритмічно підібраних прев'ю, а не до активного керування інформаційним простором. Структура інтерфейсу, представлена на Рисунку 1.1, базується на класичній трикомпонентній моделі: статичний хедер із глобальним пошуком, фіксована ліва панель для навігації по розділах і підписках, та адаптивна центральна сітка з відеокартками. Верхня частина контентної області містить горизонтальний стек фільтрів за категоріями, а основна видача фрагментована вбудованими блоками «Shorts», що створює жорстку, але перенасичену ієрархію елементів.

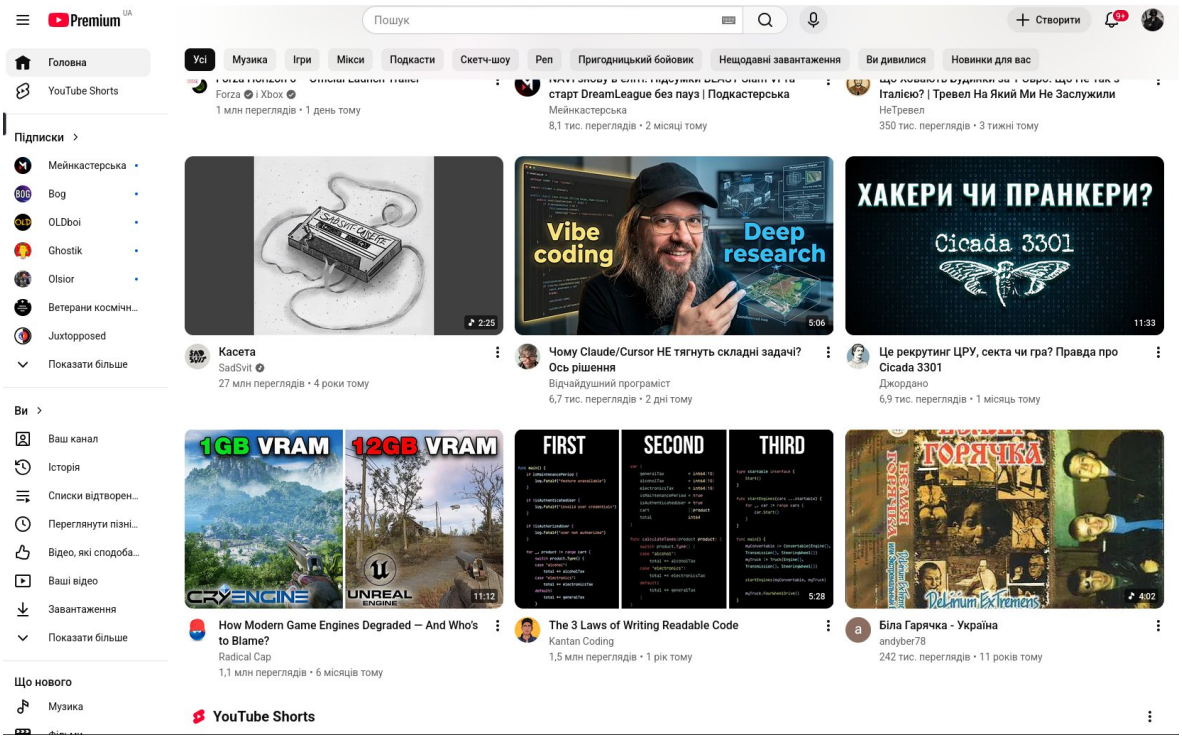

Рисунок 1.1 – Веб-інтерфейс YouTube

Наступним аналогом є Twitch, який зробив вагомий крок у бік соціальної інтерактивності. Завдяки системі розширень (Extensions), стрімери можуть додавати поверх відеопотоку функціональні оверлеї: від живої статистики

ігрових персонажів до інтерактивних карт та голосувань. Проте, попри візуальну динаміку, Twitch залишається заручником монолітної структури сторінки. Розширення на Twitch працюють як додаткові шари «поверх кадру», що часто створює хаос в інтерфейсі та не дозволяє користувачу самостійно конфігурувати робочу область. Крім того, ці інструменти є фрагментарними та рідко синхронізуються з глибокими метаданими самого відео на рівні системної логіки, що обмежує їхнє застосування в освітніх чи аналітичних сценаріях, де потрібна точність та системність. Аналогічно до попереднього представника, інтерфейс Twitch (Рисунок 1.2) базується на трикомпонентній схемі, де статичний хедер та ліва панель зі списком активних каналів формують жорсткий каркас сторінки. Проте, на відміну від YouTube, центральна область тут має складнішу ієрархію: вона поєднує інтерактивну карусель анонсів у верхній частині та тематичні блоки прямих ефірів, що підкреслює орієнтацію платформи на динамічний контент, але водночас зберігає проблему обмеженого робочого простору через надмірну кількість навігаційних елементів.

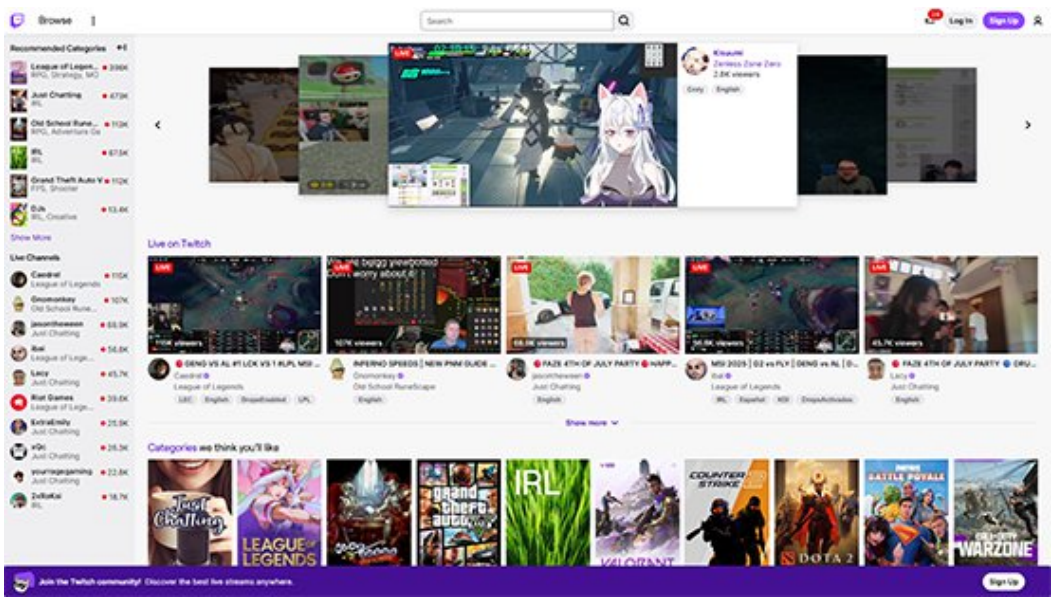

Рисунок 1.2 – Веб-інтерфейс Twitch

Третім аналогом, що представляє сучасний тренд «швидкого контенту», є TikTok. Це вершина алгоритмічного підходу, де інтерфейс максимально спроще-

ний для забезпечення безперервного потоку споживання. Проте TikTok є антиподом концепції «кокпіта». Користувач у TikTok позбавлений будь-якого контролю над структурою середовища: інтерфейс є жорстко детермінованим і незмінним. Високий рівень залученості досягається не за рахунок інструментів взаємодії, а завдяки «алгоритмічній пастці», яка робить глядача абсолютно пасивним. У TikTok відео повністю диктує умови, а користувач лише перемикає кадри, не маючи змоги отримати глибший контекст або розгорнути додаткові функціональні модулі для аналізу даних. Інтерфейс (Рисунок 1.3) зберігає традиційну модульну сітку з фіксованим лівим сайдбаром та верхньою панеллю пошуку. Проте ключовою відмінністю є перехід до вертикально-орієнтованої мозаїки контенту, де основна область заповнена ідентичними за розміром картками відео, що створює ефект нескінченної стрічки. Така структура ще більше, ніж у YouTube чи Twitch, мінімізує інтерактивні елементи навколо самого плеєра, підпорядковуючи весь візуальний простір швидкій навігації між роликами за рахунок глибини функціонального наповнення.

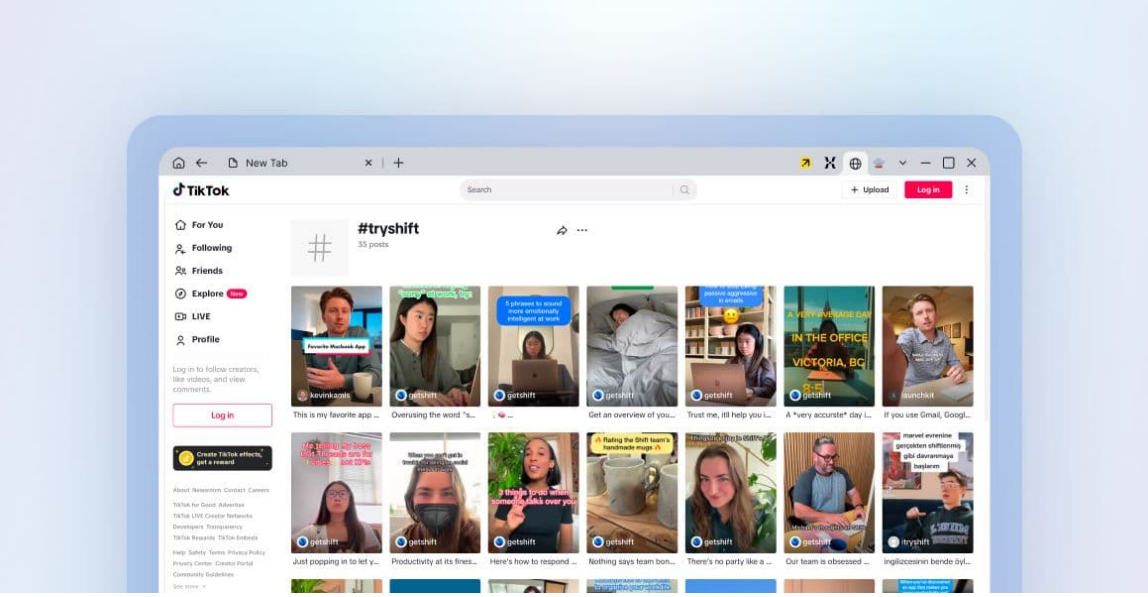

Рисунок 1.3 – Веб-інтерфейс TikTok

На основі проведеного аналізу виявлено ключові відмінності між існуючими гігантами ринку відеострімінгу та концепцією, що закладена в основу веб-

застосунку «Amplify». Головна розбіжність полягає в переході від статичного когнітивно навантаженого інтерфейсу традиційних платформ до адаптивного середовища «кокпіта».

Таблиця 1.1 – Порівняльний аналіз стратегічного позиціонування Amplify

Критерії порівняння	YouTube	Twitch	TikTok	Amplify
Роль користувача	Глядач	Соціальний учасник	Споживач алгоритму	Оператор (Пілот)
Тип інтерфейсу	Статичний моноліт	Оверлеї (шари)	Детермінований (фікс)	Динамічний кокпіт
Гнучкість UI	Відсутня	Обмежена	Нульова	Максимальна (модульна)
Мета розробки	Монетизація уваги	Соціалізація	Утримання в потоці	Ефективність роботи
Синхронізація подій	Тільки таймкоди	Ручна активізація	Відсутня	Автоматична
Архітектурна логіка	Доставка контенту	Сервісна надбудова	Алгоритмічна видача	Модульні віджети

Проведений аналіз демонструє, що сучасні медіаплатформи або зосереджені на пасивному споживанні (YouTube, TikTok), або пропонують фрагментарну інтерактивність, яка заважає перегляду (Twitch). Ринок позбавлений рішень, які б дозволяли користувачу самостійно конфігурувати свій робочий простір навколо відеопотоку. Це підтверджує актуальність розробки «Amplify» як системи, що повертає користувачу контроль над медіасередовищем.

1.3. Формування актуальності і завдань роботи

Актуальність даної роботи зумовлена глибокою суперечністю між стрімким розвитком технологій передачі медіаданих та консервативним станом інтерфейсів сучасних стрімінгових платформ. У 2026 році більшість глобальних сервісів зосереджені на алгоритмічному утриманні уваги та монетизації, що при-

звело до дефіциту функціональних інструментів для глибокої взаємодії з контентом. Традиційна модель пасивного перегляду вже не задовольняє потреби користувачів в освітній, професійній та аналітичній сферах, де відео має бути не просто зображенням, а центральним елементом динамічного робочого середовища.

Метою роботи є проектування та програмна реалізація інтерактивної стрімінгової платформи «Amplify», яка забезпечує динамічну трансформацію користувацького інтерфейсу через механізми синхронізації метаданих та мікросервісну архітектуру.

Для досягнення поставленої мети необхідно вирішити наступні завдання:

1. Провести системний аналіз предметної області, виявити функціональні обмеження існуючих медіаплатформ та обґрунтувати переваги концепції «кокпіта».
2. Обґрунтувати вибір технологічного стеку (NestJS, React, TypeScript), виходячи з вимог до масштабованості та швидкодії системи синхронізації.
3. Спроектувати архітектуру застосунку, що забезпечує незалежне функціонування та взаємодію відеопотоку з динамічними віджетами.
4. Розробити алгоритм реального часу для синхронізації подій відеопотоку (Event-to-UI sync), який дозволяє автоматично розгортати контекстні модулі навколо плеєра.
5. Реалізувати користувацький інтерфейс за модульним принципом, надавши можливість персоналізації робочого простору («кокпіта») залежно від типу медіаконтенту.

2. ПРОЄКТУВАННЯ ВЕБЗАСТОСУНКУ

2.1. Визначення користувацьких ролей і рівнів доступу до системи

Для забезпечення базових принципів інформаційної безпеки (конфіденційності, цілісності та доступності даних), а також для коректного функціонування бізнес-логіки платформи «Amplify», в архітектурі системи реалізовано модель управління доступом на основі ролей (Role-Based Access Control – RBAC).

Система проєктується за парадигмою UGC (User-Generated Content), що суттєво впливає на архітектуру доступу. У цій моделі відсутній жорсткий поділ на «виробників» та «споживачів» контенту. Кожен авторизований користувач за замовчуванням отримує гібридну роль, яка дозволяє йому одночасно споживати медіа та керувати власним інформаційним простором (каналом).

Управління правами в системі побудовано на принципі мінімальних привілеїв (Principle of Least Privilege), тобто кожен користувач отримує виключно той набір дозволів, який є критично необхідним для виконання його функцій. Уся архітектура доступу органічно поділяється на три ієрархічні рівні, які визначають глибину взаємодії з платформою.

Першим рівнем взаємодії є статус гостя – анонімного відвідувача, який ще не пройшов автентифікацію. Для таких користувачів передбачено базовий доступ виключно для читання публічного відеоконтенту та загальнодоступних сторінок. Головна мета цього рівня полягає в демонстрації можливостей платформи, зокрема відтворення HLS-потоку та перегляду роботи інтерактивних віджетів без збереження прогресу, що має стимулювати відвідувачів до реєстрації. З технічного погляду гостям суворо заборонено змінювати стан системи: блокуються будь-які запити типу POST, PUT, PATCH або DELETE, а доступ до персоналізованих стрічок чи функцій коментування залишається закритим.

Після проходження автентифікації користувач переходить на другий рівень, отримуючи базову роль зареєстрованого користувача, яка поєднує в собі

інструменти як споживача, так і автора контенту. Як глядачі, вони отримують повноцінний доступ до екосистеми: можуть залишати коментарі, формувати списки, додавати відео, ставити реакції та вести історію переглядів. Водночас їм автоматично відкривається доступ до закритого модуля Creator Studio. Тут вони перетворюються на авторів: можуть завантажувати власні медіафайли, створювати віджети та налаштовувати інтерактивні події на таймлайні, а також аналізувати залученість аудиторії через спеціальні дашборди. Проте, незважаючи на такі широкі можливості, діє суворе технічне обмеження – принцип перевірки приналежності ресурсу (Resource Ownership). Користувач може редагувати, приховувати або видаляти лише ті відео чи віджети, де його унікальний ідентифікатор (userId) збігається з ідентифікатором автора (authorId) у базі даних.

Найвищим щаблем є роль адміністратора – привілейований акаунт, призначений для власників платформи або служби підтримки. Адміністратори наділені глобальним доступом до всіх сутностей системи, незалежно від того, хто є їхнім автором. Їхня діяльність зосереджена на модерації контенту (видаленні матеріалів-порушників), управлінні обліковими записами (від попереджень до перманентних блокувань), розв'язанні технічних конфліктів та аналізі системної телеметрії. Задля безпеки робота адміністратора винесена в окремий захищений інтерфейс – Admin Panel. Крім того, на рівні бекенду всі API-запити від цієї ролі проходять обов'язкову додаткову валідацію: система перевіряє наявність спеціального маркера адміністратора безпосередньо у структурі JWT-токена.

2.2. Створення структурної схеми та діаграми прецедентів

Розробка сучасних веб-орієнтованих систем із високим навантаженням, якою є платформа для інтерактивного стрімінгу «Amplify», вимагає комплексного інженерного підходу на етапі проєктування. Перед написанням програмного коду необхідно виконати декомпозицію системи на логічні складові, що дозволяє знизити архітектурні ризики, виявити потенційні проблеми та чітко розподілити зони відповідальності між модулями.

Для забезпечення чіткого розуміння архітектури проєкту на початковому етапі було побудовано загальну структурну схему вебзастосунку (Рисунок 2.1). Вона відображає статичну організацію системи та її ключові функціональні модулі на макрорівні.

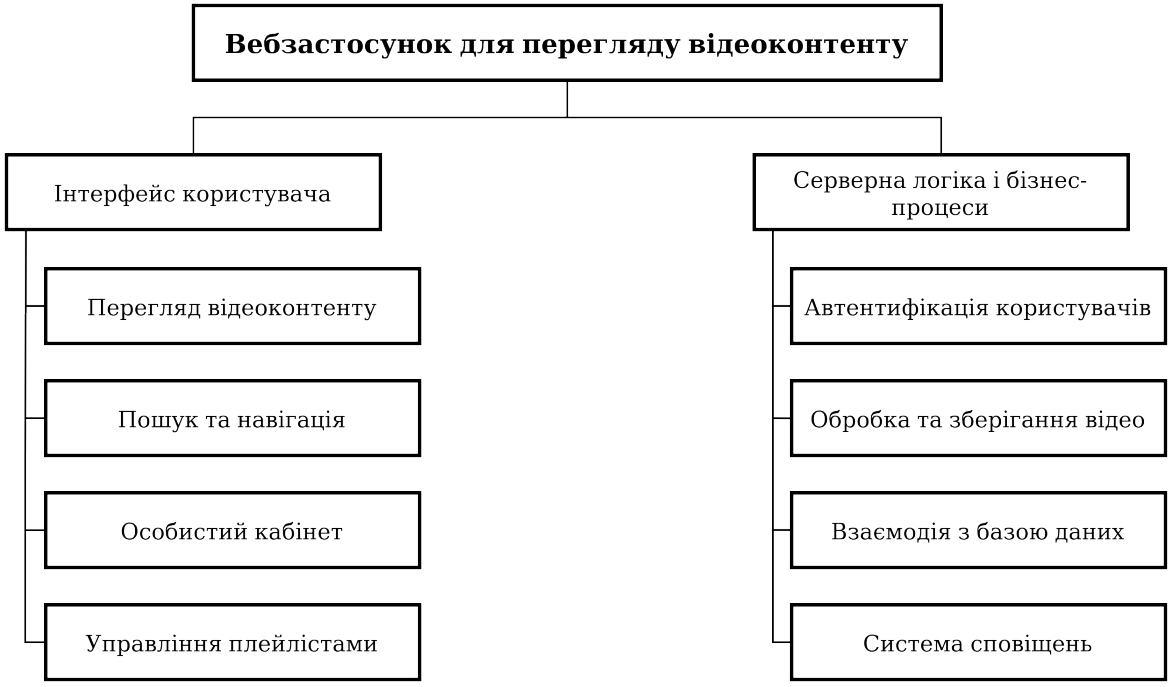

Рисунок 2.1 – Структурна схема вебзастосунку

Наведеній структурній схемі відповідає класична багаторівнева клієнт-серверна архітектура. Використання такого підходу гарантує чітке дотримання принципу єдиної відповідальності (Single Responsibility Principle) та слабку зв'язність (Loose Coupling) між інтерфейсом і бізнес-логікою. Система логічно розділена на дві великі підсистеми: «Інтерфейс користувача» та «Серверна логіка і бізнес-процеси». Для більш глибокого розуміння внутрішньої будови кожну з цих підсистем було декомпозовано окремо.

Клієнтський вебзастосунок побудований за парадигмою Single Page Application (SPA). На відміну від класичних багатосторінкових сайтів, SPA завантажує базовий HTML-документ лише один раз, а подальша навігація та оновлення даних відбуваються асинхронно через JavaScript. Це мінімізує мере-

жевий трафік, знижує навантаження на сервер і забезпечує користувачеві досвід, наближений до нативних десктопних застосунків. Детальну декомпозицію клієнтської частини наведено на Рисунку 2.2.

Рисунок 2.2 – Структурна декомпозиція клієнтської частини застосунку

Фронтенд-підсистема спроєктована з особливим фокусом на високу продуктивність рендерингу графічних елементів і об'єднує кілька взаємопов'язаних модулів у єдину цілісну екосистему. Ядром клієнтського застосунку виступає глядацький «кокпіт» – модуль перегляду відеоконтенту. Він містить інтегрований HLS-відеоплеєр на базі бібліотеки Vidstack, який працює в синергії з системою динамічного рендерингу інтерактивних віджетів, зокрема фрагментів коду CodeBlock або інтерактивних тестів Quiz. Для забезпечення плавної взаємодії ці віджети монтуються у віртуальне DOM-дерево строго синхронно з поточним часом відтворення відеопотоку.

Швидкий та ефективний доступ до всього масиву інформації забезпечує підсистема пошуку та навігації. Поряд із класичним повнотекстовим пошуком за метаданими відео, у ній реалізовано інноваційний алгоритм «глибокого пошуку» (Deep Search), який дозволяє знаходити ролики на основі текстового вмісту (payload) вбудованих у них інтерактивних віджетів. Зважаючи на UGC-природу

(User-Generated Content) платформи, модуль особистого кабінету логічно розширюється до функціоналу повноцінної Creator Studio. Ця студія пропонує авторам інтерфейс нелінійного монтажу (NLE), де за допомогою зручної технології Drag-and-Drop можна розташовувати віджети поверх або поруч із відеоплеєром, самостійно формуючи фінальний вигляд глядацького інтерфейсу. За логічне групування контенту, збереження історії переглядів та обробку соціальних реакцій – таких як підписки чи фіксація закладок на конкретних таймкодах – відповідає окремий модуль управління плейлістами.

У щільній зв'язці з клієнтським інтерфейсом функціонує серверна часть платформи, яку спроєктовано за принципами сервісно-орієнтованої архітектури (SOA). Такий архітектурний підхід дозволяє чітко розмежувати процеси обробки вхідних HTTP-запитів, внутрішні механізми захисту й безпеки даних, а також логіку безпосередньої взаємодії з компонентами хмарної інфраструктури, як це детально продемонстровано на Рисунку 2.3.

Завдяки декомпозиції бекенду на автономні функціональні блоки забезпечується висока відмовостійкість та можливість незалежного масштабування найбільш навантажених компонентів системи. Зокрема, винесення логіки обробки медіаконтенту та його збереження у хмарному сховищі дозволяє оптимізувати використання обчислювальних ресурсів під час сегментації HLS-потоків. Взаємодія між сервісами координується через оптимізовані протоколи, що мінімізує затримки при виконанні операцій та гарантує високу безпеку даних. Таке архітектурне рішення забезпечує стабільну роботу всього вебзастосунку навіть в умовах стрімкого зростання кількості одночасних користувацьких сесій.

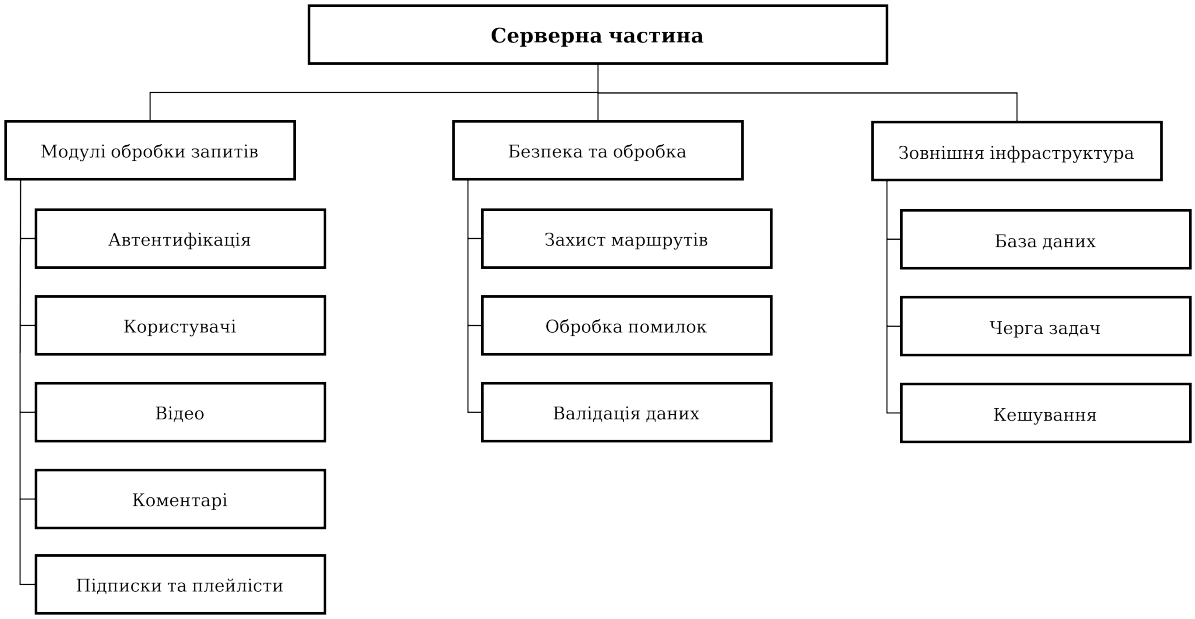

Рисунок 2.3 – Структурна декомпозиція серверної частини застосунку

Ця підсистема діє як обчислювальний центр платформи і забезпечує виконання низки ресурсомістких задач, необхідних для цілісної та стабільної роботи всіх сервісів. Основу безпеки системи становить модуль автентифікації та авторизації, який відповідає за точну ідентифікацію користувачів. У ньому реалізовано механізм Stateless-автентифікації на базі JWT-токенів (JSON Web Tokens), а також інтегровано набір захисних бар'єрів (Guards), які здійснюють сувору перевірку прав власності на ресурс (Resource Ownership) і надійно блокують будь-які спроби несанкціонованої зміни або видалення чужого контенту.

Для нівелювання високих навантажень на процесор під час виконання CPU-bound задач в архітектурі передбачено ізольований комплекс сервісів обробки та зберігання відео (Video Processing). Цей модуль бере на себе прийом вихідних файлів у форматі .mp4, їх автоматичне транскодування за допомогою інструменту FFmpeg у формат адаптивного стрімінгу HLS, а також подальше завантаження медіасегментів у хмарне об'єктне сховище (Object Storage). Надійну роботу з інформаційними потоками забезпечує прошарок абстракції для взаємодії з базою даних (Data Access Layer). Він управляє метаданими відеоро-

ликів, текстовим вмістом віджетів і зберігає зв'язки між сутностями, паралельно захищаючи БД від SQL-ін'єкцій та забезпечуючи сувору валідацію вхідних об'єктів передачі даних (DTO – Data Transfer Objects).

Асинхронну взаємодію та комунікацію всередині інфраструктури координує система сповіщень та черг задач. Вона використовує брокери повідомлень (зокрема Redis) для ефективного управління чергами транскодування відео, а завдяки протоколу WebSockets підтримує стабільний двосторонній зв'язок між клієнтом і сервером у реальному часі, що дозволяє миттєво синхронізувати стан інтерактивних віджетів та забезпечувати доставку сповіщень.

Описані вище структурні схеми відображають статичну архітектуру проєкту. Для переходу до моделювання динамічної поведінки системи, виявлення функціональних вимог та детального опису взаємодії користувачів із платформою було розроблено UML-діаграму прецедентів (Use Case Diagram), яку наведено на Рисунку 2.4.

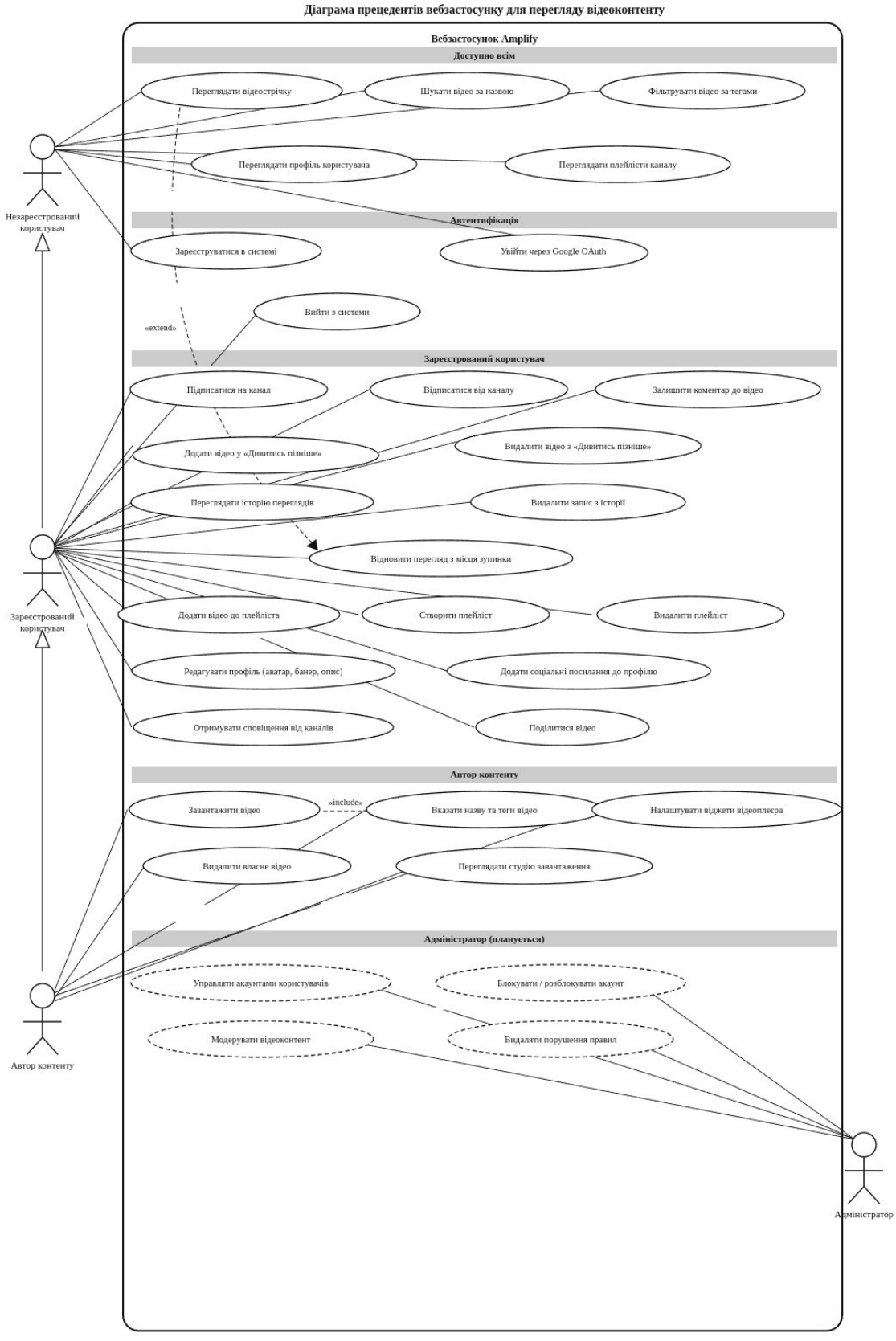

Рисунок 2.4 – Діаграма прецедентів вебзастосунку Amplify

Діаграма визначає межі системи та структуровано описує дії, які можуть виконувати зовнішні сутності (актори). Важливою архітектурною особливістю цієї моделі є використання відношення узагальнення між акторами, що ілюструє ієрархію прав доступу за принципом накопичення привілеїв.

2.3. Розробка функціональної діаграми

Для формалізації бізнес-процесів та детального опису функціональних вимог до платформи «Amplify» було використано методологію структурного аналізу та проєктування (SADT), зокрема стандарт IDEF0. На початковому етапі функціонального моделювання побудовано контекстну діаграму (рівень A0), яка відображає систему як єдиний цілісний блок, що взаємодіє із зовнішнім середовищем. Ця діаграма дозволяє чітко визначити загальні межі проєкту, вхідні та вихідні потоки інформації, механізми реалізації, а також керівні правила, яким підпорядковується робота застосунку (Рисунок 2.5).

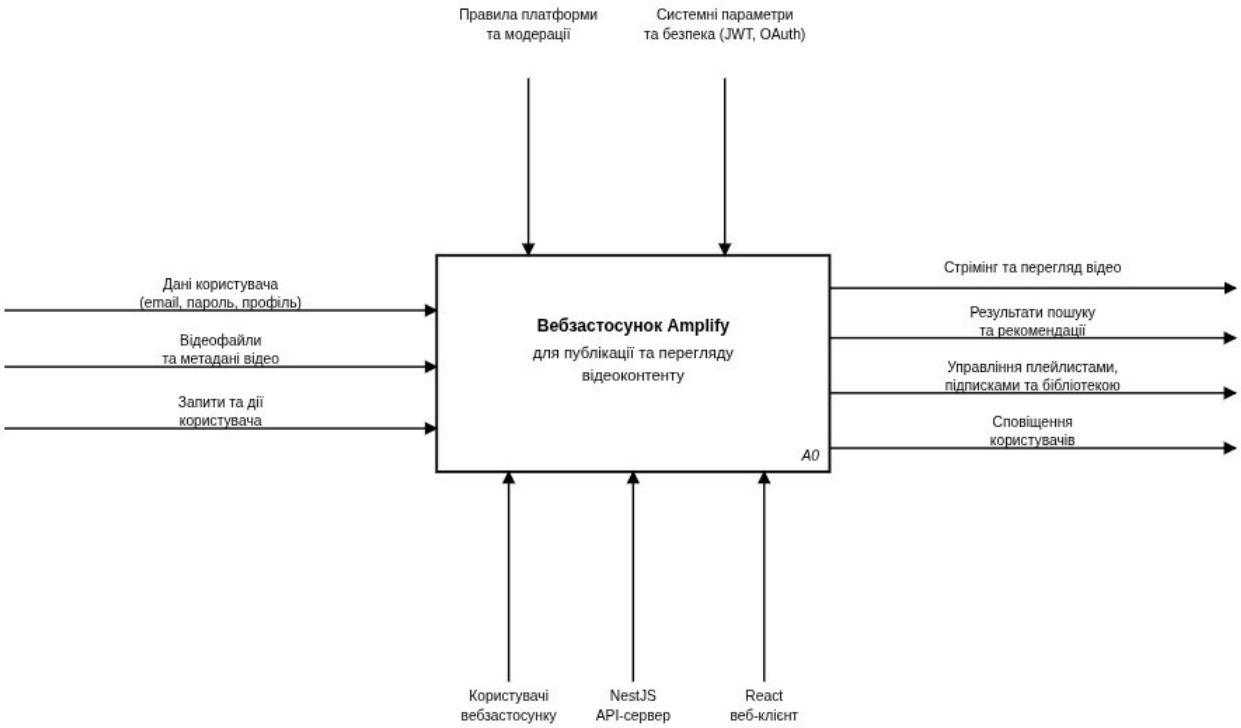

Рисунок 2.5 – Контекстна діаграма (IDEF0) вебзастосунку Amplify

Головним об'єктом моделювання на контекстній діаграмі виступає комплексний процес – «Вебзастосунок Amplify для публікації та перегляду ві-

деоконтенту». Цей центральний блок інкапсулює в собі всі внутрішні алгоритми обробки медіа, рендерингу інтерактивних віджетів та забезпечення соціальної взаємодії. З лівого боку до функціонального блоку надходять вхідні дані, які система має трансформувати або обробити. До цієї категорії належать реєстраційні та автентифікаційні дані користувача, такі як електронна пошта, пароль та інформація профілю. Також на вхід подаються безпосередньо вихідні відеофайли разом із метаданими (назвами, тегами, конфігураціями віджетів) та різноманітні запити й дії користувача, що ініціюють навігацію або зміну стану системи.

Процес перетворення вхідних даних на кінцевий результат не є хаотичним; він жорстко регламентується керуючими потоками, які входять у функціональний блок зверху. Перш за все, робота системи обмежується правилами платформи та політиками модерації, які визначають допустимість завантаженого контенту та логіку блокування порушників. Другим важливим керуючим фактором виступають системні параметри та вимоги безпеки, що включають алгоритми валідації JWT-токенів та стандарти протоколу OAuth для надійної й безпечної авторизації користувачів.

Виконання всіх функцій системи забезпечується відповідними механізмами, які підведені до головного блоку знизу. До ресурсів, що забезпечують функціонування платформи, належать безпосередньо користувачі вебзастосунку як фізичні ініціатори процесів, а також ключові технологічні компоненти архітектури: клієнтська частина на базі React (веб-клієнт) та бекенд-інфраструктура, реалізована за допомогою фреймворку NestJS (API-сервер). Ці складові спільно забезпечують логічну та апаратну реалізацію платформи.

Результатом роботи системи є вихідні дані, що генеруються на основі обробки вхідних потоків і відображаються з правого боку діаграми. Головним цільовим результатом є забезпечення безперебійного стрімінгу та перегляду відеоконтенту глядачами. Окрім цього, система формує релевантні результати пошуку та персоналізовані рекомендації контенту. Важливим вихідним

компонентом є також забезпечення функціоналу управління плейлістами, підписками та персональною бібліотекою користувача. Додатково платформа генерує системні сповіщення, які інформують авторів і глядачів про статуси обробки відеороликів, нові коментарі або активність на їхніх каналах.

Для деталізації внутрішньої логіки роботи та розкриття механізмів перетворення вхідних даних на вихідні, головний блок контекстної діаграми було декомпозовано. Результатом цього етапу стала діаграма декомпозиції першого рівня, яка розбиває складний комплексний процес на п'ять послідовних підпроцесів, тісно пов'язаних між собою інформаційними та керуючими потоками. Ця модель ілюструє поетапний життєвий цикл контенту та користувача в системі.

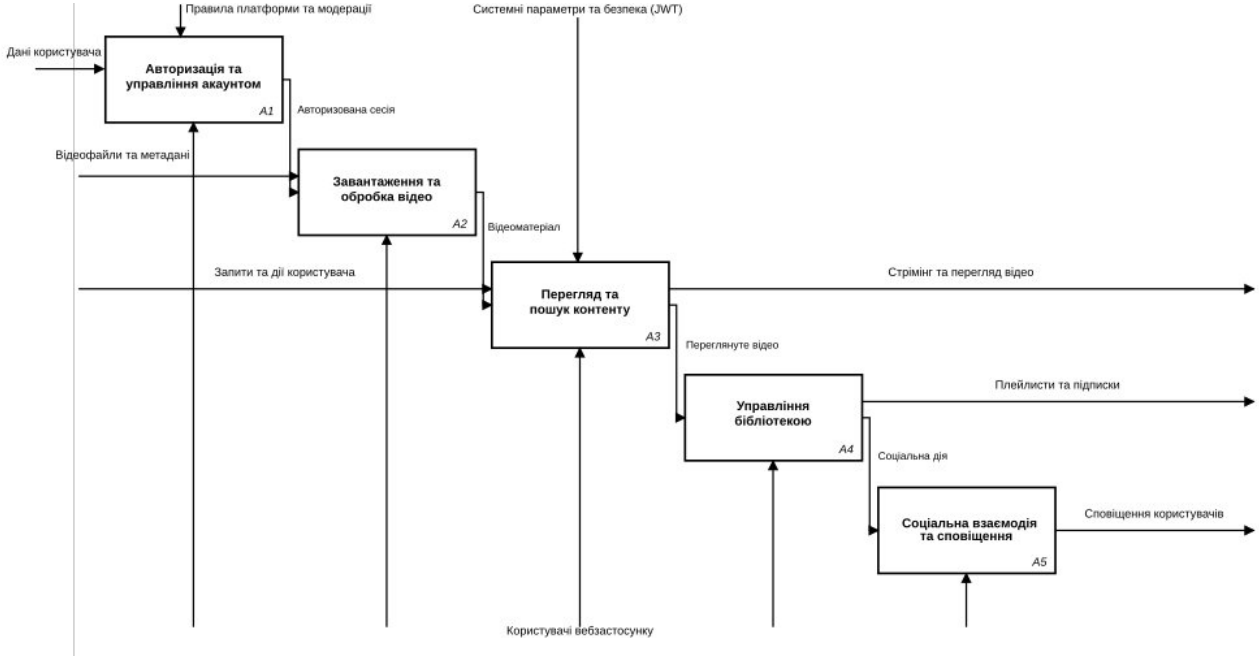

Рисунок 2.6 – Діаграма першого рівня декомпозиції

Відповідно до принципів низхідного проєктування (Top-Down Design), логічним продовженням розробки функціональної моделі є поглиблений аналіз клієнтського досвіду. На Рисунку 2.7 наведено діаграму декомпозиції рівня A3, яка розкриває внутрішні механізми підсистеми споживання контенту. Ця схема деталізує шлях користувача від формування запиту до безпосереднього перегляду медіафайлу з подальшим залученням до екосистеми платформи.

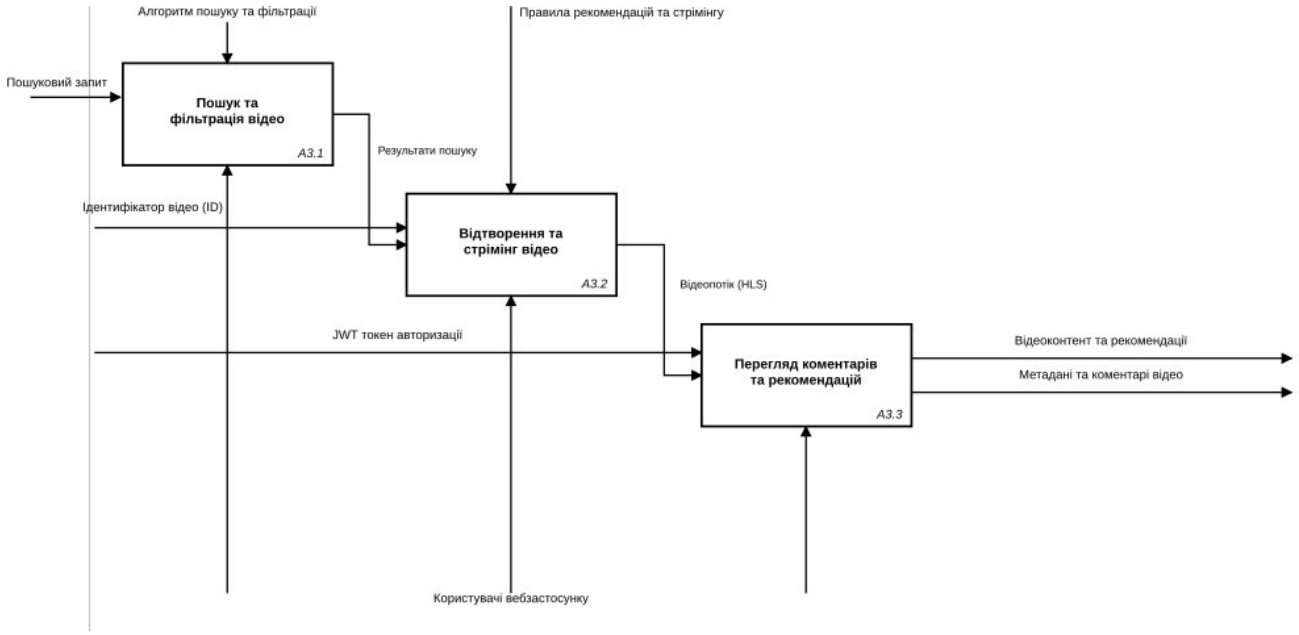

Рисунок 2.7 – Діаграма декомпозиції підпроцесу споживання відеоконтенту (рівень A3)

2.4. Створення діаграми послідовностей

Для детального аналізу було обрано один із найбільш технічно складних та ресурсомістких бізнес-процесів платформи «Amplify» – процес завантаження, асинхронної обробки та публікації відеоконтенту. Цей процес вимагає взаємодії відразу кількох ізольованих підсистем і не може бути виконаний синхронно через тривалий час конвертації медіафайлів.

У змодельованій взаємодії беруть участь шість ключових об'єктів, що представляють життєві лінії системи. Ініціатором усього процесу виступає користувач (автор), чия початкова взаємодія відбувається через клієнтську частину додатку – інтерфейс (SPA). Цей фронтенд-компонент приймає дії користувача та формує відповідні звернення до бекенду.

Наступною ланкою є сервер (API), який відіграє роль головного шлюзу для прийому та первинної обробки всіх вхідних запитів. Отримавши дані від клієнта, сервер взаємодіє з базою даних, яка слугує основним сховищем для фіксації поточних станів системи та збереження необхідних метаданих.

Оскільки процес передбачає виконання важких операцій, які не повинні блокувати роботу основного API, архітектура передбачає використання черги (BullMQ). Вона виконує функцію брокера повідомлень для надійного управління фоновими задачами та розподілу навантаження. Безпосереднє виконання цих ресурсоємних задач, зокрема транскодування медіафайлів за допомогою інструментарію FFmpeg, бере на себе обробник відео (Worker) – повністю ізольований сервіс, що працює паралельно з основним додатком.

Взаємодію поділено на два можливі сценарії: базовий (успішний) та альтернативний (обробка помилки) та представлено на Рисунку 2.8.

Розглядаючи сценарій успішного завантаження та публікації відео, процес починається з того, що користувач відкриває сторінку студії, заповнює форму з назвою та тегами, після чого ініціює завантаження файлу. Клієнтський застосунок відправляє POST-запит із multipart-даними та JWT-токеном на API сервера. Замість того, щоб блокувати з'єднання до повного завершення обробки відео, сервер діє асинхронно. Спочатку він створює новий запис у базі даних зі статусом PROCESSING і отримує його унікальний ідентифікатор. Після цього сервер відправляє задачу з параметрами ідентифікатора та шляху до файлу до брокера повідомлень BullMQ і миттєво повертає клієнту статус успішного створення (201 Created). Це дозволяє клієнтському інтерфейсу розблокуватися і демонструвати користувачеві прогрес виконання операції.

На наступному етапі в роботу вступає ізольований обробник відео, який забирає задачу з черги та починає конвертацію вихідного файлу у формат адаптивного стрімінгу HLS, генеруючи сегменти та головний маніфест. Після успішної конвертації цей обробник самостійно оновлює запис у базі даних, змінюючи статус відео на READY та записуючи його тривалість разом із URL-посиланням. На завершення система ініціює створення сповіщень для підписників каналу та звітує брокеру BullMQ про успішне виконання поточної задачі.

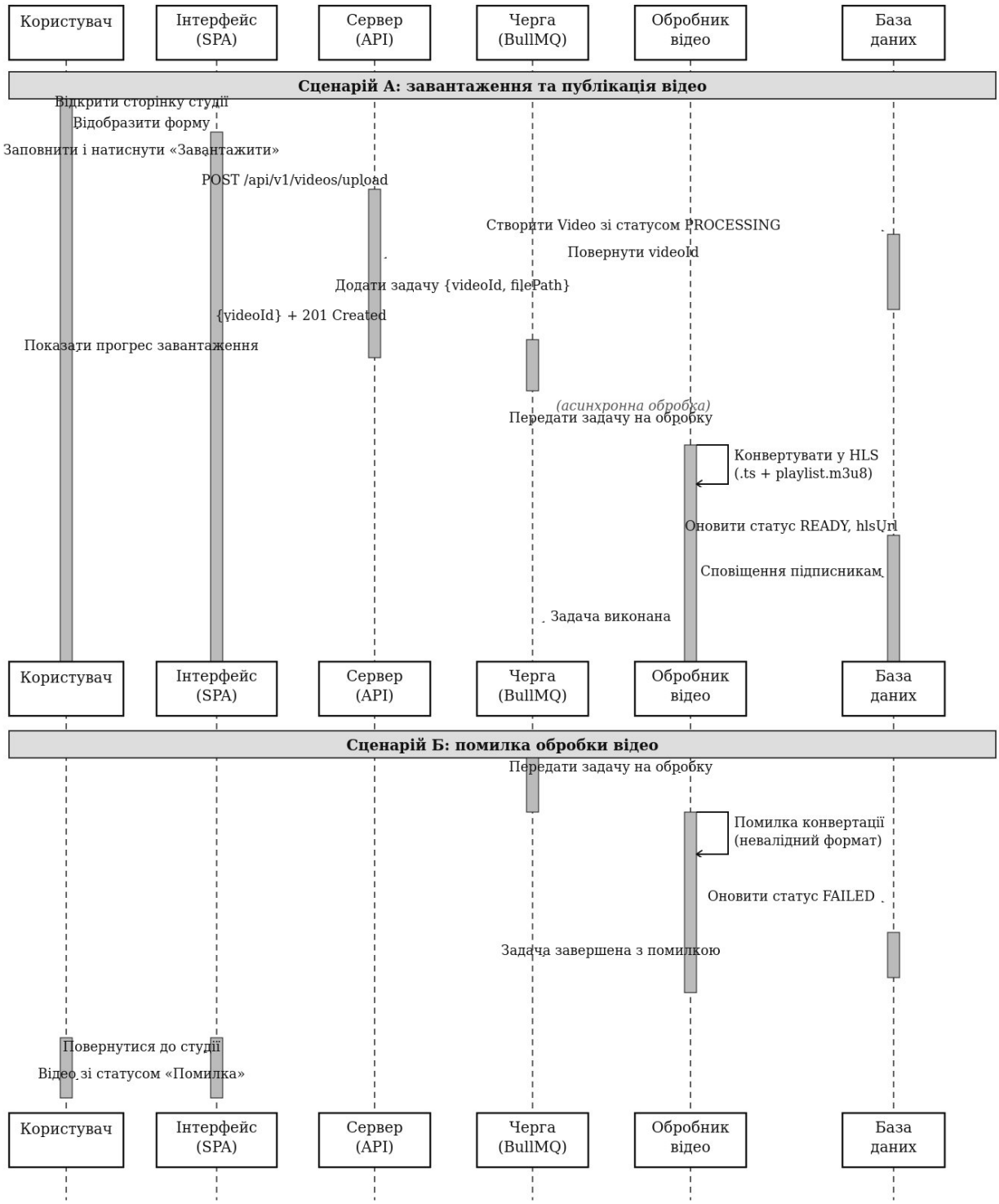

Рисунок 2.8 – Діаграма послідовностей завантаження відео у створюваному застосунку

Альтернативним сценарієм є виникнення помилки під час обробки медіафайлу. У випадку, якщо завантажене відео виявилося пошкодженим або має

непідтримуваний формат, процес транскодування переривається. Обробник відео перехоплює цю виняткову ситуацію та оновлює статус відео у базі даних на FAILED, після чого відповідна задача в черзі позначається як неуспішна. Коли користувач пізніше повертається до своєї студії через клієнтський застосунок, він бачить статус помилки біля свого відео і має змогу повторити спробу завантаження.

З архітектурної точки зору побудова цієї діаграми наочно демонструє переваги обраного патерну проєктування. Використання черги задач та делегування обчислювально важких операцій окремому обробнику відео гарантує високу відмовостійкість усієї системи. Завдяки такому підходу, навіть при пікових навантаженнях або масовому завантаженні контенту, головний сервер API залишатиметься доступним для інших користувачів, а клієнтський інтерфейс не зависатиме в очікуванні відповіді від сервера.

2.5. Побудова моделі «сутність-зв’язок» бази даних вебзастосунку

Процес проектування архітектури бази даних для розроблюваного вебзастосунку розпочався з формування логічної моделі «сутність-зв’язок», яка виступає фундаментальним інструментом для забезпечення цілісності, консистентності та масштабованості системи в умовах високих навантажень. В основу проектування покладено реляційну парадигму, що дозволяє чітко декомпозувати предметну область на ізольовані зони відповідальності: автентифікацію, медіаконтент, соціальні взаємодії та аналітику. Центральною ланкою моделі виступає сутність користувача (users), яка через розгалужену систему зовнішніх ключів інтегрована з кожним функціональним модулем, що забезпечує не лише персоналізацію клієнтського досвіду, а й суворе дотримання прав доступу до об'єктів системи. Загальна структура взаємозв'язків між таблицями представлена на Рисунку 2.9.

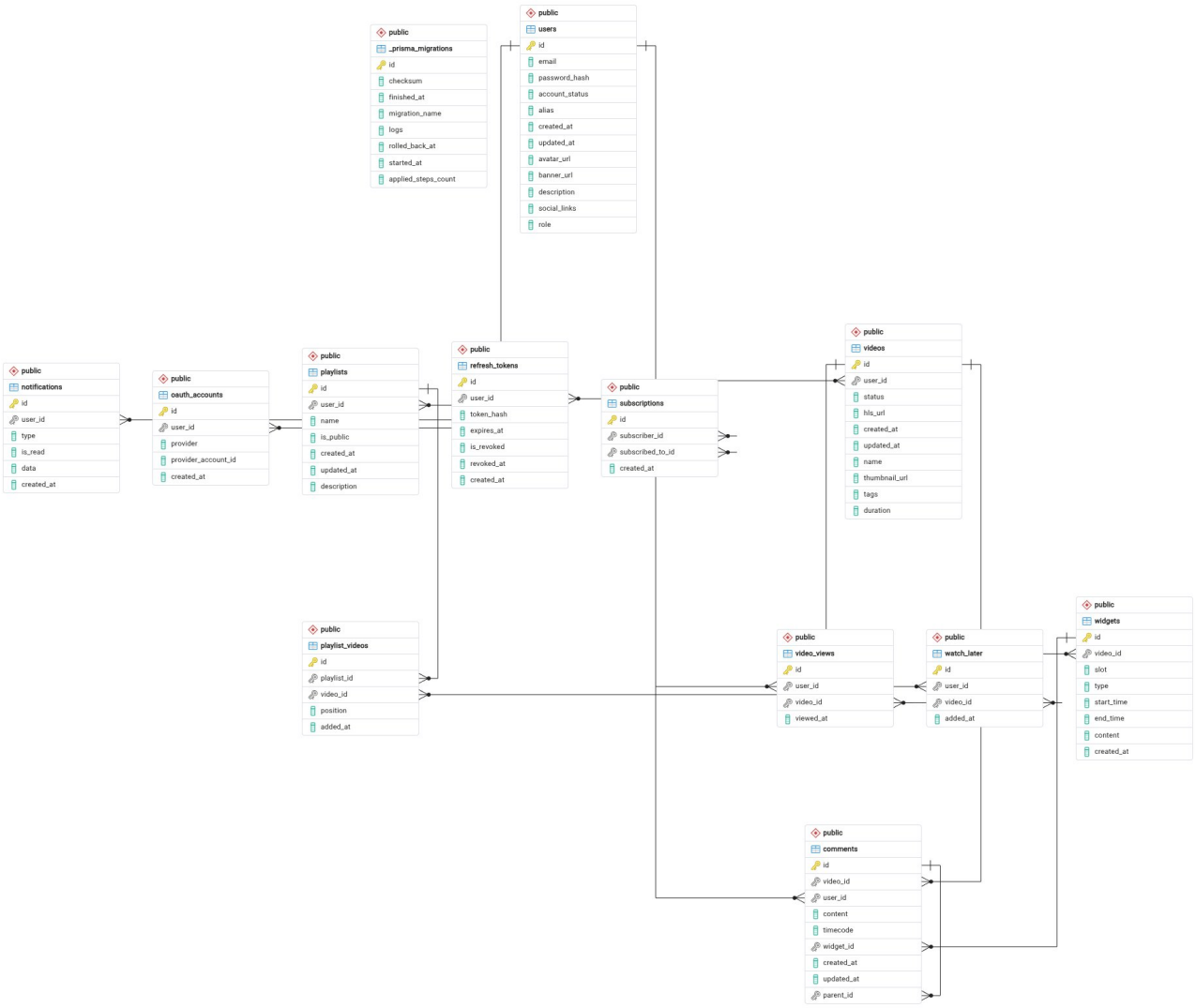

Рисунок 2.9 – Схема моделі «сутність-зв’язок» бази даних вебзастосунку

Детальний аналіз архітектури контенту дозволяє виділити сутність відео (videos) як найбільш складний об'єкт системи. Окрім стандартних атрибутів, таких як назва та опис, модель враховує специфіку стрімінгового відтворення через зберігання посилань на HLS-плейлисти та метаданих тривалості. Для забезпечення високої продуктивності запитів при великій кількості переглядів, статистика винесена в окрему сутність video_views, що дозволяє проводити глибоку аналітику поведінки користувачів без блокування основної таблиці з відео. Окрему увагу при проектуванні приділено системі коментарів (comments). На відміну від спрощених лінійних моделей, тут реалізовано рекурсивний зв’язок типу «один-

до-багатьох» через атрибут parent_id. Таке рішення дозволяє формувати складні деревоподібні структури обговорень, що сприяє підвищенню інтерактивності платформи.

Соціальний шар вебзастосунку базується на реалізації гнучких механізмів підписок та групування контенту. Сутність підписок (subscriptions) реалізує складний логічний зв'язок між двома екземплярами таблиці користувачів, розділяючи ролі на «підписника» та «автора». Це дозволяє ефективно будувати запити для генерації персоналізованих стрічок новин. Організація медіатеки через плейлисти (playlists) використовує класичну схему «багато-до-багатьох» із впровадженням проміжної сутності playlist_videos. Дана таблиця є не просто сполучною ланкою, а несе додаткове семантичне навантаження, зберігаючи порядковий номер відео в черзі відтворення, що дозволяє користувачеві кастомізувати структуру своїх списків.

Важливим технічним рішенням стало використання гібридного підходу до зберігання даних за допомогою типу JSONB. У сутностях notifications та users це дозволяє зберігати неструктуровані дані, такі як налаштування профілю або контекстні повідомлення, без необхідності проведення ресурсомістких операцій з модифікації схеми (ALTER TABLE). Такий підхід у поєднанні з використанням сучасних ORM-інструментів (Prisma) забезпечує гнучкість розробки та швидку адаптацію бази даних під нові бізнес-вимоги. Автентифікаційний рівень винесено в окремі сутності oauth_accounts та refresh_tokens, що гарантує високий рівень безпеки та дозволяє підключати сторонні провайдери ідентифікації без порушення нормалізації основних таблиць. Представлена модель відповідає вимогам третьої нормальної форми, що виключає дублювання даних та гарантує надійність збереження інформації при виконанні транзакцій.

3. ПРОГРАМНА РЕАЛІЗАЦІЯ МОДУЛІВ ПРОЕКТУ

3.1. **Вимоги до системи та базовий архітектурний підхід**

Проєктування архітектури медіа-платформи «Amplify» розпочалося з детального аналізу специфічних вимог, зумовлених її форматом Software as a Service (SaaS). Оскільки ядром системи є робота з мультимедійним контентом, першочерговою вимогою виступає забезпечення високої пропускної здатності та відмовостійкості під час інтенсивних операцій введення/виведення (I/O). Система повинна ефективно обробляти великі обсяги відеоданих, здійснюючи їх асинхронне транскодування у формат адаптивного стрімінгу (HLS). Цей процес вимагає суворої ізоляції ресурсоємних задач від основного потоку обробки клієнтських запитів, що диктує необхідність застосування масштабованих архітектурних патернів із використанням фонових черг [5].

Окрім базового функціоналу відеохостингу, унікальною рисою платформи є наявність інтерактивного шару – системи динамічних віджетів (вікторин, опитувань, термінальних блоків), які жорстко прив'язані до конкретних таймкодів відеороликів. Ця особливість формує другу критичну вимогу: необхідність підтримки надійної двосторонньої комунікації в реальному часі. Архітектура повинна гарантувати мінімальну затримку (latency) при синхронізації станів між клієнтом та сервером під час спільних переглядів або масових взаємодій з віджетами. Водночас платформа містить розгалужену бізнес-логіку управління користувачами, підписками, коментарями, плейлистами та історією активності. Такий обсяг функціоналу вимагає високого рівня абстракції та строгих контрактів між компонентами системи для уникнення структурної деградації (архітектурної ентропії) у процесі тривалої розробки.

Для задоволення вимог щодо надійності та консистентності даних фундаментальним архітектурним рішенням став вибір TypeScript як єдиної мови програмування для всього технологічного стеку. На відміну від динамічно типізованого JavaScript, TypeScript запроваджує строгу статичну типізацію на етапі

компіляції. Емпіричні дослідження доводять, що використання статичної типізації дозволяє виявляти до 15% критичних помилок та логічних вразливостей ще до запуску коду в середовищі виконання (runtime) [6]. В контексті платформи «Amplify» використання ізоморфного (універсального) коду дозволяє створити єдине джерело істини для всіх моделей даних. Завдяки цьому інтерфейси відповідей серверного API (Data Transfer Objects) та схеми валідації ідеально синхронізуються з клієнтськими моделями управління станом.

Зважаючи на необхідність тісної інтеграції серверної та клієнтської частин, а також для ефективного управління спільною кодовою базою, було застосовано архітектуру монорепозиторію (monorepo). Цей підхід передбачає консолідацію кількох ізольованих, але логічно пов'язаних підпроєктів у єдиному репозиторії системи контролю версій, що значно спрощує управління наскрізними залежностями [7]. Структура проєкту чітко розподілена на функціональні застосунки (серверне API та Web-клієнт) та набір розділюваних пакетів, які містять спільні типи, конфігурації лінтерів, компоненти користувацького інтерфейсу та утиліти. Такий архітектурний дизайн вирішує класичну проблему синхронізації версій: будь-яка зміна в моделі бази даних на сервері автоматично ініціює перевірку типів у клієнтському додатку.

Для технічного забезпечення роботи монорепозиторію було обрано зв'язку пакетного менеджера pnpm та інструменту оркестрації Turborepo. Використання pnpm з його механізмом робочих просторів («workspaces») забезпечує строгу ізоляцію залежностей через систему символічних посилань (symlinks), усуваючи проблему фантомних модулів та оптимізуючи використання дискового простору [8]. Зі свого боку, Turborepo бере на себе роль інтелектуального планувальника завдань. Шляхом аналізу графа залежностей проєкту він забезпечує паралельне виконання скриптів (збірка, тестування, перевірка типів) та запроваджує потужний механізм кешування [9]. Якщо вихідний код певного модуля не зазнав змін, інструмент пропускає його повторну обробку, миттєво повертаючи попередній

результат. В умовах розгалуженої системи це рішення стало критично важливим для забезпечення високої швидкості циклів безперервної інтеграції (CI/CD) та підтримки комфортного середовища розробки.

3.2. Серверна інфраструктура та управління даними

Проєктування серверного сегмента платформи вимагало побудови стійкої, модульної та високопродуктивної архітектури. Серед популярних рішень екосистеми Node.js (Express, Fastify, NestJS) вибір було зупинено на фреймворку NestJS 10. На відміну від мінімалістичних аналогів, NestJS пропонує строгу архітектурну матрицю, засновану на принципах об'єктно-орієнтованого програмування та механізмі впровадження залежностей (Dependency Injection) [10]. Це дозволило декомпозувати серверну частину на понад десять ізольованих функціональних модулів (автентифікація, управління відео, віджети, підписки тощо) без архітектурної деградації. Безпека платформи реалізована через механізми Guards та Interceptors із застосуванням JWT-токенів та OAuth 2.0, а для синхронізації контрактів із клієнтом використовується автоматично згенерована OpenAPI документація (Swagger).

Управління персистентним станом платформи покладено на реляційну СУБД PostgreSQL 16. Враховуючи складну логічну структуру системи, яка налічує 13 пов'язаних сутностей, реляційна модель гарантує повну підтримку вимог ACID, що є критичним для забезпечення цілісності даних при паралельних транзакціях [11]. Для взаємодії із СКБД застосовано сучасне об'єктно-реляційне відображення Prisma 6. Замість класичної декларації моделей через класи, Prisma використовує декларативну схему, на основі якої генерується суворо типізований клієнт (Prisma Client). Це гарантує цілковиту type-safe взаємодію з базою даних на рівні TypeScript та надає надійний інструментарій для міграцій [12].

Критичним інфраструктурним викликом для SaaS-відеохостингу є транскодування медіафайлів у формат HLS за допомогою утиліти fluent-ffmpeg. Оскільки ця задача є ресурсоємною (CPU-bound) і здатна повністю заблокувати

однопотоковий цикл подій (Event Loop) середовища Node.js, було імплементовано асинхронну модель обробки. Завдяки інтеграції системи черг BullMQ 5 та in-memory брокера повідомлень Redis 7, сервер негайно повертає клієнту відповідь про успішне завантаження [13]. Водночас ізольовані фонові процеси (workers) виконують кодування відео, контролюють прогрес та забезпечують автоматичне відновлення після збоїв.

Для забезпечення ізоляції інфраструктурних компонентів та загальної консистентності середовища розробки застосовано технологію контейнеризації на базі Docker Compose. Оркестрація мінімалістичних контейнерів (PostgreSQL 16 та Redis 7 на базі Alpine Linux) гарантує ідентичність версій програмного забезпечення у всіх розробників та усуває проблеми несумісності локальних конфігурацій [14]. Зі свого боку, серверний застосунок оптимізовано для надшвидкої збірки завдяки сучасному компілятору SWC (Speedy Web Compiler) в інфраструктурі Webpack 5.

3.3. Клієнтська архітектура та користувацький інтерфейс

Реалізація клієнтського сегмента медіа-платформи «Amplify» (apps/web) вимагала створення гнучкого, стійкого до високих навантажень на стороні браузера та повністю типізованого інтерфейсу. Центральним елементом клієнтської частини було обрано бібліотеку React 19. Використання її нової архітектури дозволило оптимізувати рендеринг компонентів за рахунок удосконаленої обробки асинхронних станів та вбудованої підтримки серверних компонентів [15]. Для збірки додатка та забезпечення швидкого циклу розробки було застосовано інструмент Vite 7. На відміну від застарілих рішень на базі Webpack, Vite використовує нативні ES-модулі браузера під час локального запуску, що забезпечує миттєве гаряче оновлення модулів (Hot Module Replacement) незалежно від масштабів проєкту [16]. З метою забезпечення пошукової оптимізації (SEO) сторінок перегляду відео та пришвидшення початкового завантаження інтерфейсу було впроваджено мета-фреймворк TanStack Start у

зв'язці із сервером Nitro. Це рішення дозволило реалізувати серверний рендеринг (SSR) без прив'язки до інфраструктури окремих хмарних провайдерів, що є типовим обмеженням для Next.js.

Критично важливим завданням під час побудови Single Page Application (SPA) з розгалуженою системою сторінок та динамічних шляхів стала організація маршрутизації. У проєкті було імплементовано TanStack Router 1.132, який став основою для створення файлової структури роутингу. Головною перевагою цього інструменту є забезпечення стовідсоткової безпеки типів (Type-Safe Routing) на рівні TypeScript [17]. Усі параметри шляху (наприклад, ідентифікатор відео в /video/$videoId) та query-параметри пошуку типізуються автоматично. Це дозволило повністю виключити появу битих посилань та помилок навігації ще на етапі компіляції. Для управління глобальним станом додатку було застосовано бібліотеку Zustand 5. Порівняльний аналіз із Redux Toolkit показав, що Zustand має значно менший обсяг службового коду (boilerplate) та не потребує обгортання додатку в контекст-провайдери. У проєкті було організовано три ізольованих сховища даних: для фіксації сесії користувача, динамічного стану відеоплеєра (включаючи таймкоди віджетів) та метаданих користувацького інтерфейсу [18].

Візуальне оформлення та верстка інтерфейсу виконані за допомогою фреймворку Tailwind CSS 4 із використанням підходу Utility-First. Інтеграція нової версії через плагін @tailwindcss/vite дозволила здійснювати компіляцію стилів безпосередньо в процесі збірки, мінімізуючи підсумковий розмір CSS-файлів [19]. На базі Tailwind CSS було розгорнуто компонентну бібліотеку shadcn/ui, яка використовує низькорівневі примітиви Radix UI та Base UI. Це забезпечило відповідність інтерфейсу стандартам доступності (W3C WAI-ARIA) та дозволило швидко кастомізувати складні елементи управління студії відео. Специфіка редактора інтерактивних віджетів вимагала інтеграції додаткових інструментів: механіку перетягування елементів реалізовано через бібліотеку dnd-kit, а можливість динамічної зміни розмірів та положення блоків поверх ві-

деопотоку забезпечено за допомогою модуля react-rnd. Для відображення навчального коду у відповідних віджетах інтегровано інструмент підсвічування синтаксису Shiki 4.

3.4. Технології обробки мультимедіа та синхронізації в реальному часі

Центральним функціональним ядром медіа-платформи «Amplify» є підсистема відтворення відео з накладеним шаром інтерактивності. Традиційний підхід до доставки медіаконтенту через завантаження цілісних файлів у форматі MP4 (Progressive Download) є технічно застарілим і не відповідає сучасним вимогам до відеохостингів. Він не дозволяє динамічно адаптувати якість відео під поточну пропускну здатність інтернет-з'єднання користувача. Для вирішення цієї проблеми було впроваджено протокол HTTP Live Streaming (HLS), який є галузевим стандартом для адаптивного стрімінгу [20]. На стороні сервера за допомогою утиліти fluent-ffmpeg оригінальний відеофайл сегментується на короткі фрагменти (chunks) тривалістю кілька секунд та транскодується у декілька профілів розширення (1080p, 720p, 480p) із генерацією відповідних плейлистів .m3u8.

Для відтворення HLS-потоку на клієнтській стороні застосовано зв'язку бібліотеки hls.js 1.6 та UI-фреймворку Vidstack (@vidstack/react). Бібліотека hls.js забезпечує нативну підтримку протоколу в браузерах, які не мають вбудованої реалізації HLS, керуючи буферизацією та автоматичним перемиканням бітрейту. Зі свого боку, Vidstack виступає гнучкою, доступною (a11y) оболонкою плеєра [21]. Глибока інтеграція API Vidstack з архітектурою React дозволила реалізувати ключову інновацію проєкту – механізм високоточного накладання інтерактивних віджетів поверх відеокадру із жорсткою прив'язкою до поточного таймкоду (currentTime).

Зберігання згенерованих мультимедійних сегментів, обкладинок та оригінальних файлів вимагає використання надійного хмарного об'єктного сховища. Де-факто стандартом індустрії є Amazon S3, проте його ціноутворення перед-

бачає значні витрати на вихідний трафік (egress fees). Для SaaS-платформи з інтенсивним споживанням відеоданих це може призвести до неконтрольованого зростання операційних витрат. З метою економічної оптимізації було обрано хмарне сховище Cloudflare R2 [22]. Його фундаментальною перевагою є повна відсутність тарифікації за вихідний трафік при збереженні повної сумісності з S3 API. Це дозволило безперешкодно використати офіційний клієнт @aws-sdk/client-s3 у серверному додатку NestJS для завантаження медіафайлів, забезпечивши enterprise-рівень надійності зберігання даних за значно нижчою вартістю обслуговування.

Синхронізація інтерактивних елементів системи вимагає підтримання постійного двостороннього зв'язку між клієнтом та сервером. Оскільки такі інструменти як опитування або вікторини потребують миттєвого оновлення стану для всіх глядачів, традиційні методи HTTP-запитів (Long Polling) або односторонні Server-Sent Events (SSE) не забезпечують потрібної гнучкості та створюють надмірне навантаження на сервер. Для розв'язання цього завдання застосовано протокол WebSockets через імплементацію бібліотеки Socket.IO 4 (модуль @nestjs/websockets на сервері та socket.io-client на клієнті). Ключовим аргументом на користь Socket.IO стала вбудована підтримка механізму ізольованих каналів («кімнат» / rooms) та просторів імен (namespaces) [23]. Під час відкриття сторінки конкретного відео користувач автоматично приєднується до відповідної WebSocket-кімнати. Це дозволяє серверу здійснювати мультиадресне розсилання (multicast) подій (наприклад, надходження нового коментаря або зміна результатів вікторини) виключно тим клієнтам, які в даний момент переглядають цей відеоролик, ізолюючи цей трафік від глобальної системи сповіщень платформи.

4. ВИКОРИСТАННЯ РОЗРОБЛЕНОГО ЗАСТОСУНКУ

4.1. Загальний огляд інтерфейсу та навігація

Розроблений веб-застосунок пропонує користувачам сучасний, ергономічний та інтуїтивно зрозумілий інтерфейс, побудований за принципом єдиного макета. Архітектура візуальної частини розроблена з урахуванням вимог до мінімізації когнітивного навантаження під час взаємодії з платформою, що дозволяє користувачеві швидко орієнтуватися в системі та отримувати доступ до ключових функцій. Структурно графічний інтерфейс поділяється на кілька статичних та динамічних зон, які забезпечують безперервність користувацького досвіду під час навігації між різними сторінками застосунку.

Верхня панель виступає головним якірним елементом керування, який залишається видимим на будь-якому етапі роботи з системою. У лівій частині хедера розміщено логотип платформи, який одночасно слугує посиланням для швидкого повернення на головну сторінку. Центральну частину займає багатофункціональний рядок глобального пошуку. Цей інструмент дозволяє здійснювати наскрізний пошук контенту за назвами відеороликів, закріпленими тегами та іменами авторів каналів. З метою оптимізації навантаження на серверну частину та уникнення надсилання зайвих запитів під час швидкого введення тексту, алгоритм пошуку використовує механізм дебаунсу із затримкою у 400 мілісекунд. Відповідно, система ініціює пошуковий процес і перенаправляє користувача на сторінку результатів лише після того, як він завершить введення або натисне клавішу введення. У правій частині хедера розташовані елементи персоналізації та комунікації: кнопка перемикання візуальної теми між світлим та темним режимами, іконка дзвіночка для доступу до панелі сповіщень та випадаюче меню управління обліковим записом (Рисунок 4.1).

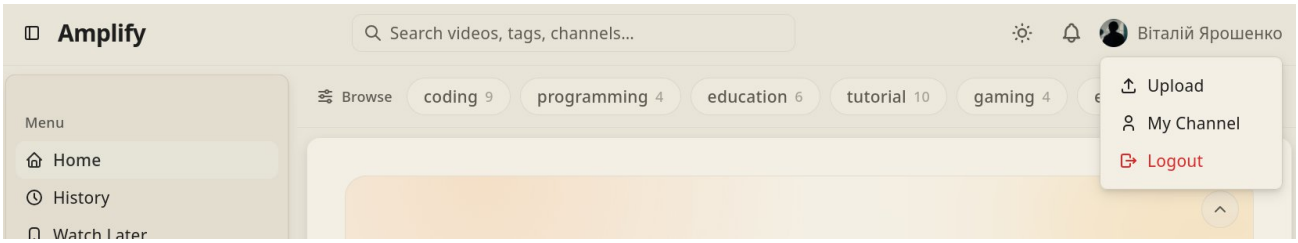

Рисунок 4.1 – Структура верхньої навігаційної панелі платформи «Amplify»

Основна робоча область застосунку динамічно змінює свій вміст залежно від обраного маршруту. При першому відвідуванні платформи або переході на головну сторінку користувач потрапляє у простір, спроєктований для максимального залучення уваги. У верхній частині контентної зони розташовується інформаційний банер, який розкриває основну концепцію інтерактивного стрімінгу та пропонує дві швидкі дії у вигляді кнопок: «Почати перегляд» для переходу до популярного контенту та «Завантажити відео» для переходу в студію автора. Зважаючи на те, що банер займає значну частину екранного простору, користувач має змогу згорнути його, звільнивши місце для відеоматеріалів.

Безпосередньо під інформаційним блоком розміщується система горизонтальної фільтрації контенту. Вона реалізована у вигляді панелі з категоріями, такими як ігри, освіта, музика, технології, спорт та інші. Кожна категорія супроводжується візуальним лічильником, який інформує про кількість доступних відеороликів за даним тегом. Натискання на будь-яку з категорій миттєво фільтрує загальну стрічку, залишаючи лише релевантний контент.

Сама стрічка відеоматеріалів побудована за принципом адаптивної сітки, яка здатна гнучко масштабуватися, відображаючи від однієї колонки на смартфонах до п'яти колонок на широких моніторах (Рисунок 4.2). Для забезпечення безперервного споживання контенту реалізовано механізм нескінченного прокручування. Замість класичної посторінкової пагінації, яка вимагає від користувача додаткових кліків, система автоматично підвантажує наступну групу з двадцяти відеороликів, щойно скрол наближається до нижньої межі сторінки. Важливим UX-рішенням на цьому етапі є використання скелетон-станів. У мо-

менти, коли клієнтський застосунок очікує відповіді від сервера з новими даними, замість порожнього екрана або стандартного індикатора завантаження, система відображає сірі анімовані блоки, які повторюють форму майбутніх карток відео. Це створює психологічний ефект швидшої роботи застосунку та зменшує рівень відмов через очікування. Логічним завершенням взаємодії з головною сторінкою є клік на картку обраного відео, після чого система здійснює маршрутизацію до ключового компонента платформи – сторінки перегляду з інтерактивним кокпітом.

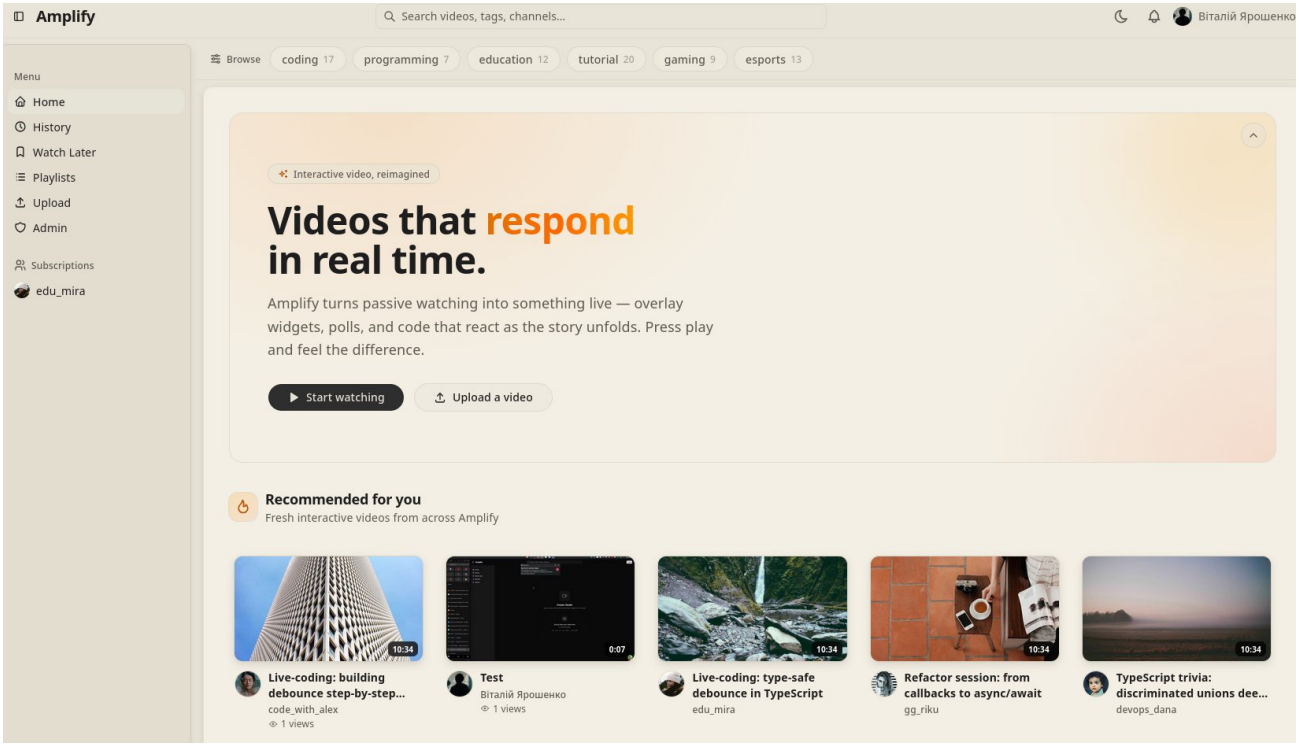

Рисунок 4.2 – Головна сторінка з інформаційним банером та адаптивною сіткою контенту

4.2. Реєстрація, авторизація та управління профілем

Для забезпечення високого рівня безпеки та зручності користувачів у вебзастосунку «Amplify» реалізовано сучасну систему автентифікації на базі протоколу Google OAuth 2.0. Відмова від традиційної форми реєстрації з ручним введенням електронної пошти та генерацією пароля дозволила суттєво знизити поріг входу для нових користувачів, зводячи процес створення облікового запи-

су до кількох кліків. Ініціація процесу входу здійснюється натисканням кнопки авторизації, що розташована у правій частині верхньої навігаційної панелі. Після цього система викликає стандартне модальне вікно провайдера Google, де користувач підтверджує надання базових прав доступу до свого профілю.

Успішна авторизація миттєво відображається у візуальному інтерфейсі платформи. На місці кнопки входу у хедері з'являється мініатюра аватара користувача. Натискання на неї відкриває контекстне меню облікового запису, яке надає швидкий доступ до ключових персональних розділів: студії завантаження контенту, сторінки власного каналу та опції безпечного виходу із системи. Архітектура застосунку чітко розмежовує потоки взаємодії для гостьових та авторизованих користувачів, використовуючи механізм захищених маршрутів екранування доступу.

Неавторизований відвідувач має змогу переглядати головну сторінку, здійснювати пошукові запити та безпосередньо дивитися відеоролики. Проте доступ до персоналізованих функцій – таких як історія переглядів, списки відкладеного перегляду, створення особистих плейлистів, керування підписками та публікація коментарів – вимагає наявності активної сесії. У разі спроби скористатися цим функціоналом без авторизації, система застосовує механізм м'якого обмеження: замість видачі технічної помилки користувач отримує контекстне повідомлення з пропозицією увійти в систему.

Після первинного створення облікового запису кожен користувач автоматично отримує власну сторінку профілю, яка одночасно виконує роль його публічного каналу. Перехід до цього розділу здійснюється за відповідним маршрутом, що містить унікальний ідентифікатор автора. Структурно простір профілю поділено на візуальну інформаційну панель (хедер каналу) та зону контенту з логічними вкладками. У верхній частині відображається ключова статистика та ідентифікатори: графічний аватар, унікальний псевдонім (alias), актуальна кількість підписників та дата реєстрації на платформі. Для сторонніх

відвідувачів каналу в цій зоні також розміщується інтерактивна кнопка підписки або відписки. Важливим ергономічним рішенням є використання принципу оптимістичного оновлення інтерфейсу під час взаємодії з цією кнопкою: її стан та лічильник підписників змінюються миттєво після натискання, не чекаючи підтвердження від сервера. Це формує у користувача відчуття максимальної швидкодії та безперебійності роботи системи.

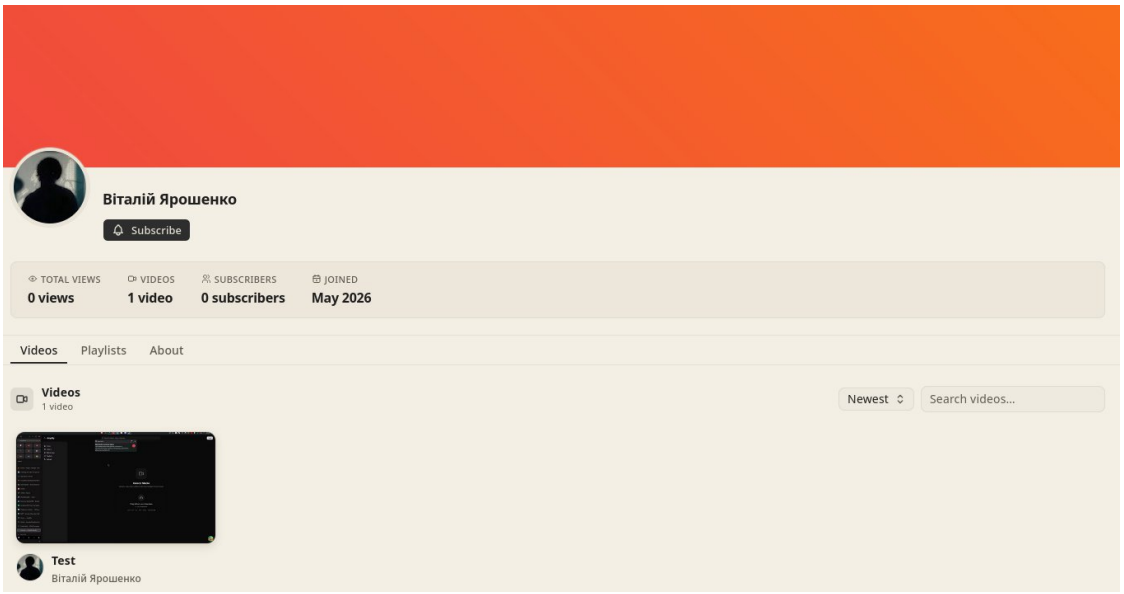

Рисунок 4.3 – Структура публічної сторінки каналу користувача

4.3. Перегляд відео та інтерактивні віджети

Центральним та найбільш інноваційним компонентом розробленої платформи «Amplify» є сторінка перегляду відеоконтенту. Одразу після переходу користувача за маршрутом обраного відео, система автоматично згортає бічну навігаційну панель. Це ергономічне рішення дозволяє вивільнити максимальну ширину екрана для робочого простору. Архітектура сторінки побудована за принципом двоколонкового макета. Основна, тобто ліва колонка, містить безпосередньо відеоплеєр, зону нижніх інтерактивних віджетів, блок метаінформації та секцію коментарів. Права колонка, яка має фіксовану ширину від 320 до 384 пікселів, відведена під відображення активних бічних віджетів під заголовком «Live now» та список рекомендованих відеороликів (Рисунок 4.4). З метою

забезпечення адаптивності, на мобільних пристроях права колонка приховується, а всі її елементи переносяться у загальний потік.

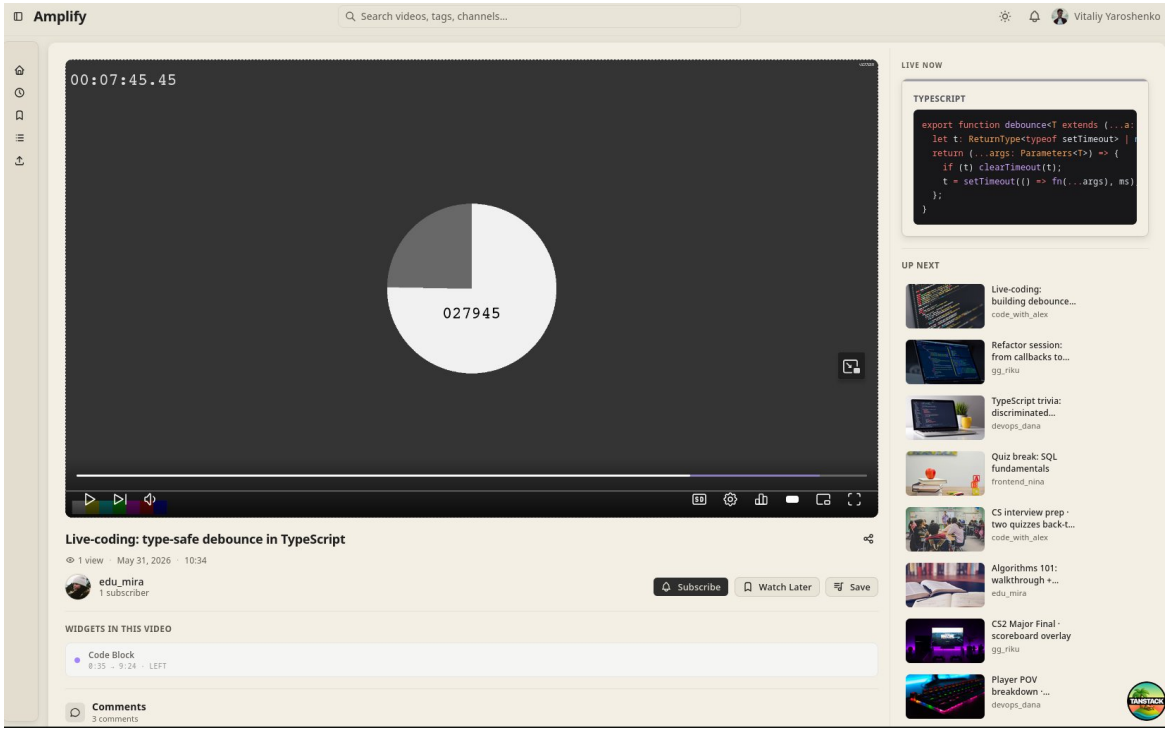

Рисунок 4.4 – Загальна структура сторінки перегляду відео

Головна функціональна особливість системи полягає у наявності інтерактивного кокпіта (Cockpit Layout). Віджети жорстко прив'язані до хронометражу відео через параметри часу початку та завершення. Вони з'являються та зникають в режимі реального часу, супроводжуючись плавними анімаціями появи та зсуву. Елементи можуть розміщуватися у двох доступних слотах: бічні віджети відображаються у правій колонці, а нижні – безпосередньо під плеєром.

Безпосередньо під відеоплеєром розташована панель метаінформації, яка містить назву відео, кількість переглядів, дату публікації, тривалість, а також дані про канал автора. Авторизований користувач може взаємодіяти з контентом за допомогою ряду кнопок: підписатися на канал, додати до списку відкладеного перегляду або зберегти у конкретний плейлист через випадаюче меню. Варто відзначити, що дія підписки використовує механізм оптимістичного оновлення,

миттєво змінюючи стан кнопки без очікування відповіді від сервера. Кнопка «Share» дозволяє швидко скопіювати посилання у буфер обміну.

Логічним завершенням робочої області є модернізована секція коментарів, яка глибоко інтегрована у контекст відео. Коли користувач встановлює фокус на поле введення тексту, система автоматично фіксує поточну секунду відтворення та виводить підпис формату «Commenting at M:SS». Якщо в цей конкретний момент на екрані відображається активний віджет, алгоритм автоматично пов'язує створюваний коментар із ним. Усі часові мітки в текстах коментарів перетворюються на інтерактивні посилання, натискання на які перемотує плеєр до відповідного епізоду. Система підтримує створення відповідей на існуючі коментарі, формуючи зручне деревоподібне обговорення. Для забезпечення захисту від спаму, гостьовим користувачам замість форми введення відображається повідомлення з пропозицією авторизуватися для отримання можливості залишати коментарі.

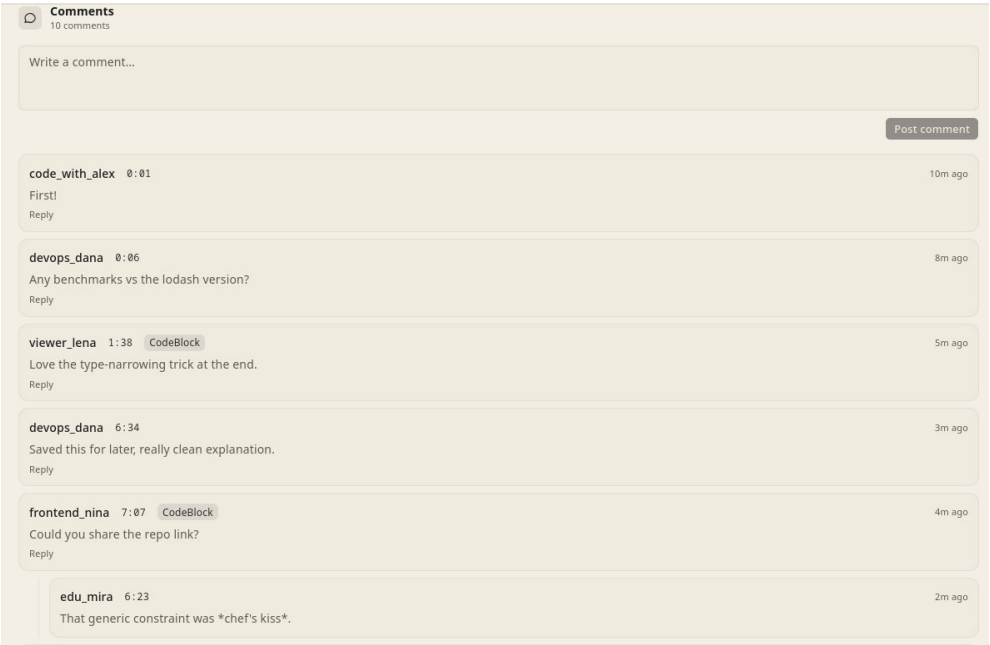

Рисунок 4.5 – Інтерактивна система коментарів із прив'язкою до хронометражу відеоролика

4.4. Завантаження та редагування відео

Для забезпечення процесу публікації контенту в застосунку реалізовано спеціалізований модуль «Creator Studio», доступний за відповідним навігаційним маршрутом. Процес публікації розпочинається з першого кроку – вибору медіа-файлу у зоні завантаження. Система підтримує широкий спектр сучасних відео-форматів, включаючи файли з розширеннями mp4, mov, avi, mkv та webm, що забезпечує високу гнучкість використання застосунку (Рисунок 4.6). Максимальний допустимий обсяг файлу становить 500 мегабайтів. З огляду на адаптивність платформи, зона завантаження динамічно змінює текстові підказки залежно від типу пристрою користувача: на настільних комп'ютерах відображається заклик перетягнути файл безпосередньо у робочу область екрана, тоді як на мобільних пристроях система пропонує торкнутися екрана для відкриття системного провідника файлів.

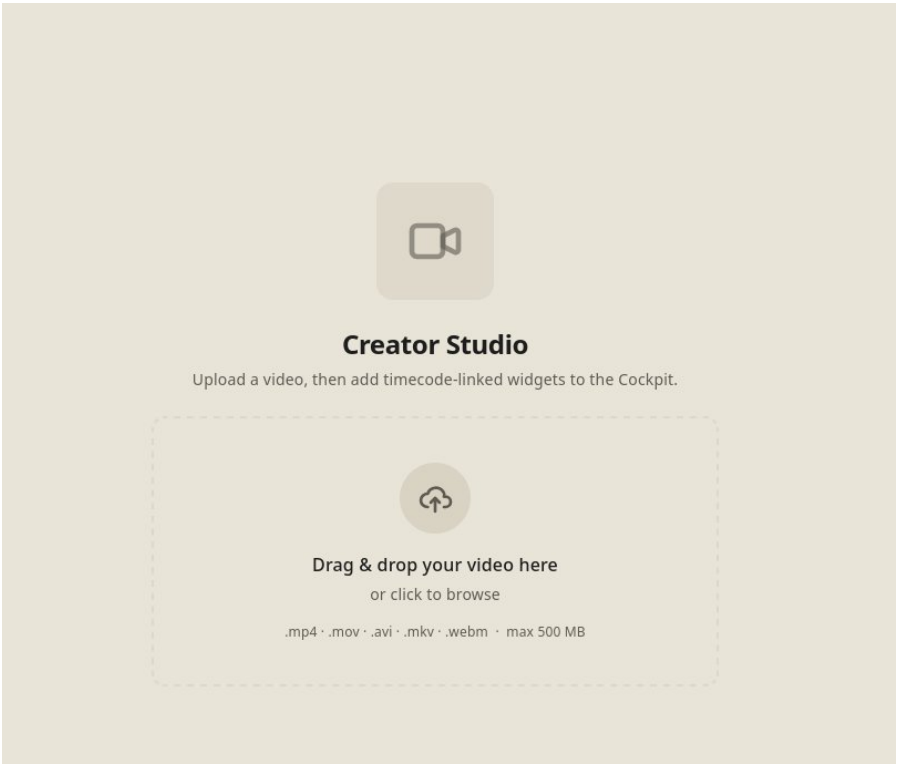

Рисунок 4.6 – Інтерфейс вибору відеофайлу для завантаження

Після успішного вибору файлу інтерфейс миттєво трансформується у повноцінний нелінійний редактор віджетів. Ця панель містить дві логічні вклад-

ки. Перша вкладка призначена для введення метаданих та вимагає від автора заповнення обов'язкового поля назви відеоролика. Тут також передбачені інструменти для додавання текстового опису обсягом до 500 символів, завантаження графічної мініатюри (із можливістю візуального попереднього перегляду та видалення) і вибору релевантних категорій через інструмент множинного вибору з відображенням візуальних бейджів. Друга вкладка слугує панеллю управління інтерактивними віджетами, де у вигляді списку виводяться всі додані до відео елементи. Вона надає автору глибокий доступ до параметрів віджета, включаючи зміну часових міток, вибір зони розміщення та можливість безпосереднього редагування структури даних (JSON-контенту) обраного елемента.

Права, значно ширша колонка, відведена під візуальний відеоредактор, який поєднує в собі полотно попереднього перегляду та часову шкалу. Полотно містить дві активні зони для розміщення інтерактивних елементів: праву та нижню. Користувач має змогу інтерактивно взаємодіяти з цими зонами, клікаючи або перетягуючи туди нові віджети з меню. Під полотном розташована панель управління відтворенням із базовими інструментами, повзунком перемотування, який підсвічується під час взаємодії, та точним лічильником часу.

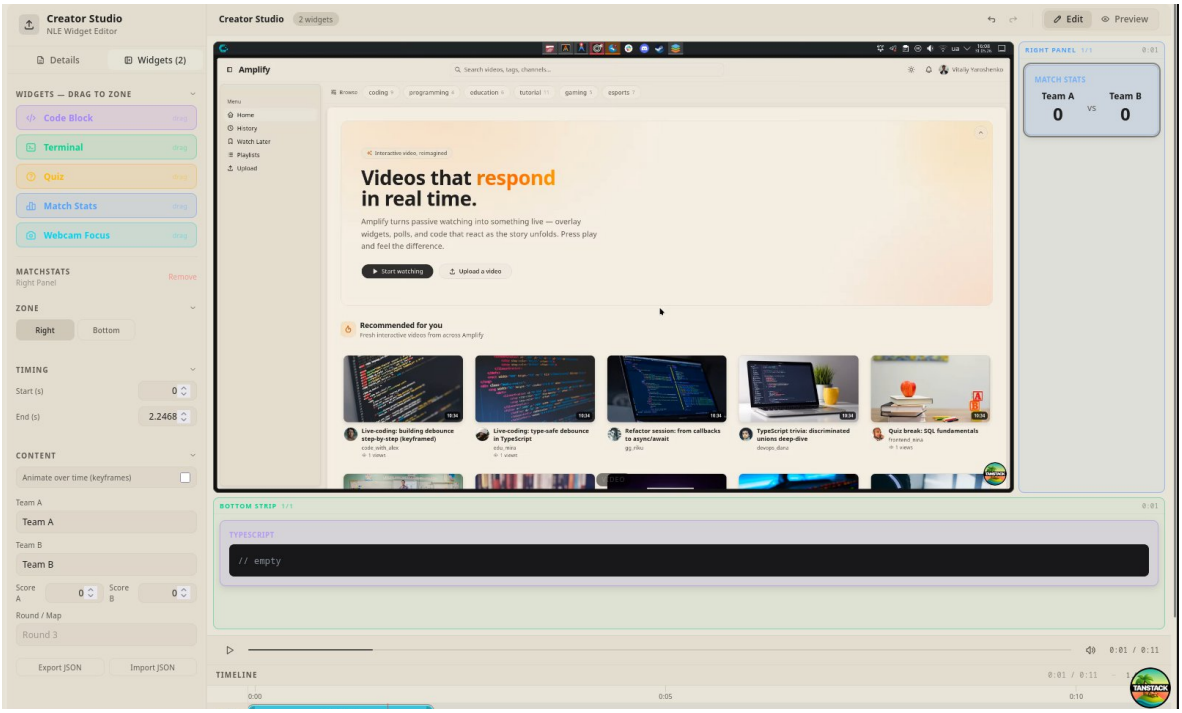

Рисунок 4.7 – Інтерфейс нелінійного редактора віджетів

Ключовим інструментом для точного позиціювання віджетів є часова шкала (Рисунок 4.8). Кожен доданий інтерактивний елемент репрезентується на ній у вигляді окремої кольорової смужки, що відповідає його типу та обраній зоні розміщення. Автор відео може інтуїтивно перетягувати ці смужки вздовж шкали для зміни часу появи віджета, виділяти їх кліком миші для подальшого налаштування або миттєво видаляти за допомогою клавіші Backspace. Для забезпечення максимального контролю над процесом створення контенту, панель інструментів редактора містить перемикач між режимами попереднього перегляду та редагування.

Рисунок 4.8 – Часова шкала

Для просунутих користувачів реалізовано можливість не лише ручного налаштування параметрів через графічний інтерфейс, але й функцію імпорту конфігурацій: автор може завантажити готовий JSON-файл із параметрами віджета. Ця функція дозволяє зовнішньо генерувати складні структури даних (наприклад, масиви коду або розгалужені тести) і миттєво імпортувати їх у систему, а також безпосередньо редагувати JSON-контент прямо в панелі.

Завершальним етапом роботи у студії є публікація матеріалу. Кнопка публікації стає активною виключно після заповнення обов'язкового поля назви відео, у разі спроби розпочати завантаження без вказання назви, система виводить відповідну попереджувальну підказку. Процес вивантаження файлу на сервер супроводжується візуальним індикатором прогресу у відсотках.

ВИСНОВОК

У межах виконаної кваліфікаційної роботи було розроблено веб-застосунок «Amplify» – інтерактивну стрімінгову платформу, що реалізує концепцію динамічного «кокпіта» для персоналізованого споживання медіаконтенту. Усі поставлені завдання було вирішено в повному обсязі, а визначена мета досягнута.

За результатами системного аналізу предметної галузі виявлено, що сучасні медіаплатформи (YouTube, Twitch, TikTok) демонструють технологічну стагнацію у розвитку користувацьких інтерфейсів. Їхні архітектурні моделі орієнтовані на пасивне споживання контенту та монетизацію уваги, тоді як інструментальна глибина для активної взаємодії з медіаматеріалом залишається вкрай обмеженою. Проведений порівняльний аналіз підтвердив актуальність розробки платформи нового типу, що повертає користувачеві роль активного оператора власного інформаційного середовища.

На етапі проєктування розроблено повний комплект архітектурної документації: структурну схему вебзастосунку з декомпозицією клієнтської та серверної підсистем, діаграму варіантів використання із визначенням трирівневої моделі доступу (RBAC), контекстну IDEF0-модель та діаграму послідовностей асинхронного завантаження і транскодування відео. Спроєктовано реляційну модель бази даних, що налічує 13 взаємопов’язаних сутностей та відповідає вимогам третьої нормальної форми.

Програмну реалізацію здійснено на базі ізоморфного TypeScript-стеку в архітектурі монорепозиторію (pnpm + Turborepo). Серверну частину побудовано на фреймворку NestJS 10 із застосуванням модульного моноліту, ORM Prisma 6 над PostgreSQL 16, системи черг BullMQ 5 на Redis 7 для асинхронного транскодування FFmpeg, а також WebSocket-протоколу (Socket.IO 4) для синхронізації інтерактивних подій у реальному часі. Клієнтська частина реалізована на React 19 із Vite 7, TanStack Router, Zustand 5 та компонентною бібліотекою

shadcn/ui. Доставку медіаконтенту забезпечено за протоколом HLS із хмарним сховищем Cloudflare R2.

Ключовою технічною інновацією застосунку є механізм Event-to-UI sync – алгоритм синхронізації поточного таймкоду відеопотоку з динамічним рендерингом інтерактивних віджетів (Quiz, CodeBlock тощо) у визначених слотах «кокпіта». Реалізований нелінійний редактор (Creator Studio) із часовою шкалою та підтримкою Drag-and-Drop забезпечує авторам повний контроль над позиціюванням та налаштуванням інтерактивних елементів на таймлайні відео.

Розроблений застосунок «Amplify» демонструє практичну реалізацію принципів сучасної SaaS-архітектури та підтверджує можливість побудови відеоплатформи нового покоління, де відео виступає центральним елементом динамічного інтерактивного середовища, а не пасивним потоком для споживання. Отримані результати мають теоретичне та прикладне значення у сфері розробки освітніх, аналітичних та корпоративних медіаплатформ.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Fuchs C. Social Media: A Critical Introduction (3rd ed.). SAGE Publications, 2021.
2. Wu T. The Attention Merchants: The Epic Scramble to Get Inside Our Heads. Alfred A. Knopf, 2016.
3. Flayelle M., Maurage P., Billieux J. Toward a qualitative understanding of binge-watching behaviors: A focus group approach. Journal of Behavioral Addictions. 2017. Vol. 6(4). P. 457-471.
4. Chi M. T. H., Wylie R. The ICAP framework: Linking cognitive engagement to active learning outcomes. Educational Psychologist. 2014. Vol. 49(4). P. 219-243.
5. Newman S. Building Microservices: Designing Fine-Grained Systems (2nd ed.). O'Reilly Media, 2021.
6. Gao Z., Bird C., Barr E. T. To type or not to type: quantifying detectable bugs in JavaScript. Proceedings of the 39th International Conference on Software Engineering (ICSE '17). 2017. P. 758–769.
7. Potvin R., Levenberg J. Why Google stores billions of lines of code in a single repository. Communications of the ACM. 2016. Vol. 59(7). P. 78-87.
8. Symlinked node_modules structure. pnpm Official Documentation. URL: https://pnpm.io/symlinked-node-modules-structure.
9. Core Concepts: Caching. Turborepo Official Documentation. URL: https://turbo.build/repo/docs/core-concepts/caching.
10. NestJS Documentation: Architecture and Building Blocks. NestJS Official Reference. URL: https://docs.nestjs.com .
11. Silberschatz A., Korth H. F., Sudarshan S. Database System Concepts (7th ed.). McGraw-Hill, 2019.
12. Prisma ORM Reference: Type-Safe Database Access. Prisma Docs. URL: https://www.prisma.io/docs.

13. Redis Architecture and In-Memory Data Structures. Redis Documentation. URL: https://redis.io/docs .
14. Docker Compose Specification and Containerization Best Practices. Docker Product Docs. URL: https://docs.docker.com/compose.
15. React v19.0: New Features and Component Architecture. React Official Blog. URL: https://react.dev/blog .
16. Vite 7.0 Build Pipeline and Internal Architecture. Vite Guide. URL: https://vite.dev/guide.
17. TanStack Router & Start: Type-Safe Applications for React. TanStack Documentation. URL: https://tanstack.com/router/v1.
18. Zustand: State Management Principles in React Applications. Zustand Reference. URL: https://zustand-demo.pmnd.rs.
19. Tailwind CSS v4.0: The Fast Utility-First CSS Engine. Tailwind Architecture Docs. URL: https://tailwindcss.com/docs.
20. HTTP Live Streaming (RFC 8216). Internet Engineering Task Force (IETF). URL: https://datatracker.ietf.org/doc/html/rfc8216.
21. Vidstack Player: Architecture and React Integration. Vidstack Documentation. URL: https://www.vidstack.io/docs/player/frameworks/react.
22. Cloudflare R2 Object Storage: S3 Compatibility and Zero Egress Fees. Cloudflare Technical Documentation. URL: https://developers.cloudflare.com/r2/.
23. Socket.IO Rooms and Namespaces: Multiplexing and Event Broadcasting. Socket.IO Official Docs. URL: https://socket.io/docs/v4/rooms/.

Додаток А

Автентифікація: ротація refresh-токенів та видача JWT

```
async refreshByRawToken(
    rawRefreshToken: string,
  ): Promise<RefreshServiceResult> {
    const tokenHash = createHash("sha256")
      .update(rawRefreshToken)
      .digest("hex");

    const stored = await this.prisma.refreshToken.findFirst({
      where: { tokenHash, isRevoked: false },
      include: { user: true },
    });

    if (!stored || stored.isRevoked || stored.expiresAt < new
Date()) {
      throw new UnauthorizedException({
        code: "REFRESH_TOKEN_INVALID",
        message: "auth.errors.sessionExpired",
      });
    }

    await this.prisma.refreshToken.update({
      where: { id: stored.id },
      data: { isRevoked: true, revokedAt: new Date() },
    });

    const payload: JwtPayload = {
      sub: stored.user.id,
      email: stored.user.email,
      accountStatus: stored.user.accountStatus,
      role: stored.user.role,
    };

    const accessToken = await this.jwtService.signAsync(payload, {
      secret: this.configService.get("app.APP_JWT_ACCESS_SECRET", {
        infer: true,
      }),
      expiresIn: this.configService.get("app.APP_JWT_ACCESS_EX-
PIRY", {
        infer: true,
      }),
    });

    const { rawRefreshToken: newRawToken, tokenHash: newTokenHash }
=
      this.generateRefreshTokenHash();
```

```
    await this.prisma.refreshToken.create({
      data: {
        userId: stored.user.id,
        tokenHash: newTokenHash,
        expiresAt: new Date(Date.now() + 30 * 24 * 60 * 60 * 1000),
      },
    });

    return { accessToken, rawRefreshToken: newRawToken };
  }
```

Додаток Б

Транскодування завантаженого відео у формат HLS (FFmpeg)

```
getDuration(inputPath: string): Promise<number | null> {
    return new Promise((resolve) => {
      ffmpeg.ffprobe(inputPath, (err, metadata) => {
        if (err || !metadata?.format?.duration) {
          resolve(null);
        } else {
          resolve(Math.round(metadata.format.duration));
        }
      });
    });
  }

  async transcodeToHls(inputPath: string, videoId: string): Prom-
ise<string> {
    const outputDir = join(TEMP_HLS_DIR, videoId);

    // Clean any stale output from a previous attempt
    await rm(outputDir, { recursive: true, force: true });
    await mkdir(outputDir, { recursive: true });

    const outputManifest = join(outputDir, "index.m3u8");

    return new Promise<string>((resolve, reject) => {
      ffmpeg(inputPath)
        .outputOptions([
          "-profile:v baseline",
          "-level 3.0",
          "-start_number 0",
          "-hls_time 10",
          "-hls_list_size 0",
          "-f hls",
        ])
        .output(outputManifest)
        .on("end", () => {
```

```
          this.logger.log(`Transcoding complete for
videoId=${videoId}`);
          resolve(outputDir);
        })
        .on("error", (err: Error) => {
          this.logger.error(
            `Transcoding failed for videoId=${videoId}: ${err.mes-
sage}`,
          );
          reject(err);
        })
        .run();
    });
  }
```

Додаток В

Фонова обробка відео у черзі BullMQ із повторними спробами

```
 async process(job: Job<VideoProcessingJobData>): Promise<void> {
    const { videoId, tempFilePath } = job.data;
    const hlsDir = join(TEMP_HLS_DIR, videoId);
    const maxAttempts = job.opts.attempts ?? 1;
    const isFinalAttempt = job.attemptsMade >= maxAttempts - 1;

    this.logger.log(
      `[${videoId}] attempt=${job.attemptsMade + 1}/${maxAttempts}
— starting`,
    );

    try {
      const [duration] = await Promise.all([
        this.transcoding.getDuration(tempFilePath),
      ]);

      await this.transcoding.transcodeToHls(tempFilePath, videoId);

      this.logger.log(`[${videoId}] Uploading HLS files to R2`);
      const hlsUrl = await this.storage.uploadHlsDirectory(hlsDir,
videoId);

      const existing = await this.prisma.video.findUnique({
        where: { id: videoId },
        select: { thumbnailUrl: true },
      });

      let thumbnailUrl: string | undefined;
      if (!existing?.thumbnailUrl) {
        try {
```

```
          this.logger.log(`[${videoId}] No thumbnail — extracting
first frame`);
          const framePath = await this.transcoding.extractFirst-
Frame(tempFilePath, videoId);
          thumbnailUrl = await this.storage.uploadThumb-
nail(framePath, videoId, "image/jpeg");
        } catch (err) {
          this.logger.warn(`[${videoId}] Auto-thumbnail failed:
${(err as Error).message}`);
        }
      }

      const updatedVideo = await this.prisma.video.update({
        where: { id: videoId },
        data: {
          status: "READY",
          hlsUrl,
          ...(thumbnailUrl ? { thumbnailUrl } : {}),
          ...(duration != null ? { duration } : {}),
        },
        select: {
          id: true,
          name: true,
          thumbnailUrl: true,
          userId: true,
          user: { select: { alias: true } },
        },
      });

      this.logger.log(`[${videoId}] Processing complete. URL:
${hlsUrl}`);

      this.notifications
        .notifySubscribersNewVideo({
          videoId: updatedVideo.id,
          videoName: updatedVideo.name,
          thumbnailUrl: updatedVideo.thumbnailUrl,
          channelId: updatedVideo.userId,
          channelAlias: updatedVideo.user.alias,
        })
        .catch((err: Error) =>
          this.logger.warn(`[${videoId}] Notification dispatch
failed: ${err.message}`),
        );

      await this.cleanup(tempFilePath, hlsDir);
    } catch (err) {
      this.logger.error(
```

```
        `[${videoId}] Failed (attempt ${job.attemptsMade +
1}/${maxAttempts}): ${(err as Error).message}`,
      );

      if (isFinalAttempt) {
        await this.prisma.video
          .update({ where: { id: videoId }, data: { status:
"FAILED" } })
          .catch(() => {});
        await this.cleanup(tempFilePath, hlsDir);
      }

      throw err;
    }
  }
```

Додаток Г

Коментарі з прив'язкою до таймкоду та побудова дерева відповідей

```
const COMMENT_SELECT = {
  id: true,
  content: true,
  timecode: true,
  widgetId: true,
  parentId: true,
  createdAt: true,
  widget: { select: { id: true, type: true } },
  user: { select: { id: true, alias: true, avatarUrl: true } },
} as const;

function mapComment(c: RawComment): CommentDto {
  return {
    id: c.id,
    content: c.content,
    timecode: c.timecode,
    widgetId: c.widgetId,
    widget: c.widget,
    author: c.user,
    createdAt: c.createdAt,
    parentId: c.parentId,
    replies: (c.replies ?? []).map((r) => mapComment(r)),
  };
}

@Injectable()
export class CommentsService {
  constructor(private readonly prisma: PrismaService) {}

  async createComment(
```

```
    videoId: string,
    userId: string,
    dto: CreateCommentDto,
  ): Promise<{ id: string }> {
    const video = await this.prisma.video.findUnique({ where: { id:
videoId } });
    if (!video) throw new NotFoundException("Video not found");

    const comment = await this.prisma.comment.create({
      data: {
        videoId,
        userId,
        content: dto.content,
        timecode: dto.timecode,
        widgetId: dto.widgetId ?? null,
        parentId: dto.parentId ?? null,
      },
    });

    return { id: comment.id };
  }

  async getComments(videoId: string): Promise<GetCommentsRe-
sponseDto> {
    const comments = await this.prisma.comment.findMany({
      where: { videoId, parentId: null },
      orderBy: { timecode: "asc" },
      select: {
        ...COMMENT_SELECT,
        replies: {
          orderBy: { createdAt: "asc" },
          select: COMMENT_SELECT,
        },
      },
    });

    return { items: comments.map((c) => mapComment(c as RawCom-
ment)) };
  }
```

Додаток Ґ

Інтерполяція ключових кадрів динамічних віджетів

```
export interface Keyframe {
  /** Seconds relative to the widget's startTime (>= 0). */
  at: number
  [key: string]: unknown
}
```

```
type KeyframeWidget = Pick<Widget, "content" | "startTime">

/** Returns well-formed keyframes sorted ascending by `at`, or null
if not keyframed. */
export function getKeyframes(content: Record<string, unknown> | un-
defined): Keyframe[] | null {
  const raw = content?.keyframes
  if (!Array.isArray(raw) || raw.length === 0) return null
  const valid = raw.filter(
    (k): k is Keyframe =>
      k != null && typeof k === "object" && typeof (k as { at?: un-
known }).at === "number",
  )
  if (valid.length === 0) return null
  return [...valid].sort((a, b) => a.at - b.at)
}

/** Whether a widget's content carries an inner keyframe timeline.
*/
export function isKeyframed(content: Record<string, unknown> | un-
defined): boolean {
  return getKeyframes(content) !== null
}

/**
 * Index (into the ascending-sorted keyframe list) of the frame ac-
tive at the
 * given absolute playback time, or -1 when the widget is not
keyframed. Before
 * the first keyframe's `at` the first frame is used, so a visible
panel always
 * has content.
 */
export function activeKeyframeIndex(widget: KeyframeWidget, cur-
rentTime: number): number {
  const kf = getKeyframes(widget.content)
  if (!kf) return -1
  const rel = Math.max(0, currentTime - widget.startTime)
  let idx = 0
  for (let i = 0; i < kf.length; i++) {
    if (kf[i]!.at <= rel) idx = i
    else break
  }
  return idx
}

/**
 * Resolves a widget's content to the payload active at `current-
Time`: shared
```

```
 * fields merged with the active keyframe (minus its `at`). Non-
keyframed
 * content is returned unchanged.
 */
export function resolveWidgetContent(
  widget: KeyframeWidget,
  currentTime: number,
): Record<string, unknown> {
  const content = widget.content ?? {}
  const kf = getKeyframes(content)
  if (!kf) return content

  const idx = activeKeyframeIndex(widget, currentTime)
  const { keyframes: _drop, ...base } = content
  const { at: _at, ...frame } = kf[idx]!
  return { ...base, ...frame }
}

const clamp = (v: number, lo: number, hi: number) => Math.max(lo,
Math.min(hi, v))

/**
 * Authoring helper: clamp each keyframe's `at` into [0, maxDura-
tion] and return
 * the list sorted ascending. Used by the editor and JSON import so
stored
 * keyframes always fall inside the widget's own window.
 */
export function normalizeKeyframes(keyframes: Keyframe[], maxDura-
tion: number): Keyframe[] {
  const hi = Math.max(0, maxDuration)
  return keyframes
    .map((k) => ({ ...k, at: clamp(Number(k.at) || 0, 0, hi) }))
    .sort((a, b) => a.at - b.at)
}
```

Додаток Д

Хук визначення активних у поточний момент віджетів за таймкодом

```
export function useActiveWidgets(widgets: Widget[]) {
const widgetsRef = useRef(widgets)
widgetsRef.current = widgets

const lastTimeRef = useRef(0)
const activeSigRef = useRef<Set<string>>(new Set())
const [activeWidgets, setActiveWidgets] = useState<Widget[]>([])

const recompute = useCallback((time: number) => {
lastTimeRef.current = time
const inWindow = widgetsRef.current.filter(
```

```
(w) => time >= w.startTime && time < w.endTime,
)
// Signature includes the active keyframe index so content changes
trigger
// a re-render; non-keyframed widgets contribute "#-1" (stable).
const next = new Set(inWindow.map((w) => `${w.id}#${ac-
tiveKeyframeIndex(w, time)}`))

if (!setsEqual(activeSigRef.current, next)) {
activeSigRef.current = next
setActiveWidgets(
inWindow.map((w) => ({ ...w, content: resolveWidgetContent(w,
time) })),
)
}
}, [])

// Recompute when widgets list changes (additions, deletions, first
load).
useEffect(() => {
recompute(lastTimeRef.current)
}, [widgets, recompute])

return { activeWidgets, onTimeUpdate: recompute }
}
```

Додаток Е

Стан студії-редактора: знімкова історія undo/redo (Zustand)

```
ushHistory: () =>
      set((s) => {
        const next = [...s.undoStack, s.elements]
        // Cap history to avoid unbounded memory growth.
        if (next.length > HISTORY_CAP) next.shift()
        return { undoStack: next, redoStack: [] }
      }),

    endInteraction: () => set({ _interactionKey: null }),

    undo: () =>
      set((s) => {
        if (s.undoStack.length === 0) return {}
        const prev = s.undoStack[s.undoStack.length - 1]
        const newUndo = s.undoStack.slice(0, -1)
        const newRedo = [...s.redoStack, s.elements]
        // Drop selection if the previously-selected element no
longer exists.
        const stillExists = s.selectedId !== null && prev.some((e)
=> e.id === s.selectedId)
        return {
```

```
          elements: prev,
          undoStack: newUndo,
          redoStack: newRedo,
          selectedId: stillExists ? s.selectedId : null,
          _interactionKey: null,
        }
      }),

    redo: () =>
      set((s) => {
        if (s.redoStack.length === 0) return {}
        const next = s.redoStack[s.redoStack.length - 1]
        const newRedo = s.redoStack.slice(0, -1)
        const newUndo = [...s.undoStack, s.elements]
        if (newUndo.length > HISTORY_CAP) newUndo.shift()
        const stillExists = s.selectedId !== null && next.some((e)
=> e.id === s.selectedId)
        return {
          elements: next,
          undoStack: newUndo,
          redoStack: newRedo,
          selectedId: stillExists ? s.selectedId : null,
          _interactionKey: null,
        }
      }),

    canUndo: () => get().undoStack.length > 0,
    canRedo: () => get().redoStack.length > 0,

    addElement: (el) => {
      // Discrete action — end any in-progress interaction and push
history.
      get().pushHistory()
      set({ _interactionKey: null })
      set((s) => ({ elements: [...s.elements, { ...el, id:
uid() }] }))
    },

    duplicateElement: (id) => {
      const src = get().elements.find((e) => e.id === id)
      if (!src) return
      get().pushHistory()
      set({ _interactionKey: null })
      set((s) => {
        const copy: SceneElement = {
          ...src,
          id: uid(),
          startTime: src.endTime,
          endTime: src.endTime + (src.endTime - src.startTime),
```

```
        }
        return { elements: [...s.elements, copy], selectedId:
copy.id }
      })
    },

    updateElement: (id, patch, interactionKey) => {
      const state = get()
      // Coalesce successive updates that share an interactionKey
(e.g. a
      // single drag/resize/typing burst) into one history entry —
capture
      // history once at the start, suppress subsequent pushes.
      if (interactionKey) {
        if (state._interactionKey !== interactionKey) {
          state.pushHistory()
          set({ _interactionKey: interactionKey })
        }
      } else {
        // Discrete update (e.g. zone change, single button click)
— always push.
        state.pushHistory()
        set({ _interactionKey: null })
      }
      set((s) => ({
        elements: s.elements.map((e) => (e.id === id ?
{ ...e, ...patch } : e)),
      }))
    },
```

Додаток Є

Типізований HTTP-клієнт: дедуплікація оновлення токена та черга повторів

```
// Shared refresh deduplication — ensures only one POST
/v1/auth/refresh is in flight at a time,
// regardless of whether the caller is SessionProvider.hydrate() or
ApiClient.handleUnauthorized().
let _refreshPromise: Promise<string> | null = null;

export function refreshTokenOnce(baseUrl: string): Promise<string>
{
  _refreshPromise ??= fetch(`${baseUrl}/v1/auth/refresh`, {
    method: "POST",
    credentials: "include",
    headers: { "Content-Type": "application/json" },
  })
    .then(async (res) => {
      const json = await res.json();
      if (!res.ok) throw ApiError.fromResponse(res.status, json);
      const token =
```

```
        (json as { data?: { accessToken?: string } }).data?.ac-
cessToken ?? "";
      setApiAuthToken(token);
      return token;
    })
    .catch((err: unknown) => {
      setApiAuthToken(null);
      globalThis.dispatchEvent(new Event("auth:session-expired"));
      throw err;
    })
    .finally(() => {
      _refreshPromise = null;
    });
  return _refreshPromise;
}
…
  private handleUnauthorized<T>(
    method: string,
    path: string,
    body: unknown,
    options?: RequestOptions,
  ): Promise<ApiResponse<T>> {
    return new Promise<ApiResponse<T>>((resolve, reject) => {
      this.refreshQueue.push({
        resolve: (newToken: string) => {
          resolve(
            this.request<T>(method, path, body, {
              ...options,
              skipRefresh: true,
              headers: {
                ...options?.headers,
                Authorization: `Bearer ${newToken}`,
              },
            }),
          );
        },
        reject,
      });

      // Only the first enqueued request initiates the refresh; the
rest await the same promise.
      if (this.refreshQueue.length === 1) {
        refreshTokenOnce(this.baseUrl)
          .then((newToken) => {
            const queue = this.refreshQueue.splice(0);
            for (const entry of queue) entry.resolve(newToken);
          })
          .catch((err: unknown) => {
            const queue = this.refreshQueue.splice(0);
```

```
            for (const entry of queue) entry.reject(err);
          });
      }
    });
  }
```