

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «магістр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення»)

на тему:

**«Дослідження методів обробки потокового відео на
базі платформи Node.js»**

Виконав студент групи ПЗМ-23-1
ФЕДЧЕНКОВ Іван Юрійович

Керівник роботи:
МОРОЗОВ Андрій Васильович

Рецензент:
ЛЕВКІВСЬКИЙ Віталій Леонідович

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

«ЗАТВЕРДЖУЮ»

Зав. кафедри інженерії
програмного забезпечення

Тетяна ВАКАЛЮК

«01» жовтня 2024 р

ЗАВДАННЯ
на кваліфікаційну роботу

Здобувач вищої освіти: **ФЕДЧЕНКОВ Іван Юрійович**

Керівник роботи: **МОРОЗОВ Андрій Васильович**

Тема роботи: **«Дослідження методів обробки потокового відео на базі платформи Node.js»**

Затверджена Наказом закладу вищої освіти від **«01» жовтня 2024 р., №520/с**

Вихідні дані для роботи: **дослідження галузі обробки відео, специфіки роботи з потоковим відео, тестування різних підходів до обробки потокового відео, збір системних метрик, зберігання результатів обробки у файл з метою подальшого аналізу, аналіз ефективності і доцільності використання різних відео кодеків.**

Керівники / консультанти з магістерської кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	Керівник/консультант	Завдання видав	Завдання прийняв
1	Морозов А.В.	01.10.2024	01.10.2024
2	Морозов А.В.	02.10.2024	02.10.2024
3	Морозов А.В.	03.10.2024	03.10.2024

РЕФЕРАТ

Кваліфікаційна робота магістра складається з опису виконаного дослідження предметної області та алгоритмів, відбору серед існуючих технологій найбільш поширених та актуальних для подальшого аналізу, розробки ПЗ, аналізу отриманих даних і пояснювальної записки як результату. Пояснювальна записка до випускної роботи містить текстову частину, викладену на 65 сторінках друкованого тексту, має 8 сторінок додатків. Список використаних джерел містить 40 найменувань і займає 5 сторінок. В роботі наведено 16 рисунків та 14 таблиць. Загальний обсяг роботи – 73 сторінки.

Метою роботи є дослідження методів обробки потокового відео на базі платформи Node.js, що включає в себе роботу з кожною з технологій в різних умовах, подальший аналіз системних метрик а також візуальний аналіз отриманих даних на предмет дефектів, графічних артефактів, інших проблем, і як результат порівняння ефективності бібліотек і кодеків при виконанні різних задач.

Під час виконання роботи була досліджена значна кількість джерел за предметною областю, зокрема історія розвитку стандартів кодування відео, існуючі бібліотеки для обробки відео і їхні можливості в контексті потокового відео, класичні підходи до кодування відео і більш прогресивні. Було обґрунтовано вибір технологій для дослідження, виконано написання програмного коду, що автоматизує роботу з відео і збирає метрики, виконано аналіз отриманих результатів і проведено порівняння ефективності.

КЛЮЧОВІ СЛОВА: ПОТОКОВЕ ВІДЕО, ТРАНСКОДИНГ, РОЗДІЛЬНА ЗДАТНІСТЬ, RTSP, H.264 NODE.JS, FFmpeg, GStreamer, CODEC.

ABSTRACT

The master's thesis consists of a description of the completed research of the subject area, algorithms,

A master's thesis consists of a description of the research performed on the subject area and algorithms, selection of the most common and relevant technologies among the existing ones for further analysis, software development, analysis of the data obtained, and an explanatory note as a result. The explanatory note to the final work contains a text part, set out on 65 pages of printed text, and has 8 pages of appendices. The list of references includes 40 titles and covers 5 pages. The work contains 16 figures and 14 tables. The total volume of the work is 73 pages.

The purpose of this work is to study methods of streaming video processing based on the Node.js platform, including working with each of the technologies in different conditions, further analysis of system metrics, as well as visual analysis of the data obtained for defects, graphic artifacts, and other problems, and as a result, comparison of the effectiveness of libraries and codecs in performing various tasks.

In the course of the work, a significant number of sources in the subject area were studied, including the history of video coding standards, existing video processing libraries and their capabilities in the context of streaming video, classical approaches to video coding and more advanced ones. The choice of technologies for the study was substantiated, program code was written to automate video processing and collect metrics, the results were analyzed, and the efficiency was compared.

KEYWORDS: STREAM VIDEO, TRANSCODING, RESOLUTION, RTSP, H.264 NODE.JS, FFMPEG, GSTREAMER, CODEC.

ЗМІСТ

РЕФЕРАТ	3
ABSTRACT	4
ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ	6
ВСТУП	7
РОЗДІЛ 1. ПОСТАНОВКА ЗАДАЧІ ТА АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ РОБОТИ	10
1.1 Огляд процесу обробки потокового відео	10
1.2. Постановка задачі.....	13
1.3. Обґрунтування мови програмування для якої буде здійснюватися дослідження.....	18
1.4. Аналіз існуючих рішень в сфері обробки потокового відео:	19
Висновки до першого розділу	21
РОЗДІЛ 2. ПРАКТИЧНА РЕАЛІЗАЦІЯ ОБРОБКИ ВІДЕО	22
2.1. Реалізація пробного транскодування відео з застосуванням обраних інструментів	22
2.2. Формування вимог і проектування тестового середовища	27
2.3. Створення тестового середовища	28
Висновки до другого розділу	35
РОЗДІЛ 3. ОТРИМАННЯ ДАНИХ І АНАЛІЗ ЕФЕКТИВНОСТІ ОБРОБКИ ПОТОКОВОГО ВІДЕО	37
3.1 Робота з тестовим середовищем	37
3.2 Аналіз отриманих даних.....	37
Висновки до третього розділу.....	58
ВИСНОВКИ	59
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	61
ДОДАТКИ	66

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

FPS – frames per second, кількість кадрів на секунду

ШІ – штучний інтелект

TS – TypeScript

MJPEG – Motion-JPEG

ЦП – центральний процесор

ВСТУП

Актуальність теми. Потокове відео вже давно є ваговою частиною інформаційних технологій. Однак донедавна потреба в його обробці зростала досить лінійно, через що питання пошуку найбільш ефективних методів його обробки стояло не надто гостро. Але останніми роками можна спостерігати стрибок ефективності, а відповідно й популярності штучного інтелекту, і на даний момент не існує підстав вважати що використання штучного інтелекту сповільниться, або, тим паче, зупиниться повністю.

Але коли мова заходить про штучний інтелект то не надто часто згадують технології збирання, обробки і передачі даних для їх подальшої обробки алгоритмами штучного інтелекту. Одним зі значних напрямів розвитку штучного інтелекту став так званий комп'ютерний зір. Даний напрям фокусується на обробці візуальних даних з метою розпізнавання корисної інформації (рукописний текст, дорожні знаки і розмітка, обличчя, автомобілі, продукти і т.д.). І якщо деякі з даних систем працюють зі статичними зображеннями, то деяким навпаки, треба в простішому випадку відео, а в складнішому-потокове відео [1], тобто пряма трансляція, у разі необхідності мінімальної затримки в роботі.

Нині велика частина успішного проекту з застосуванням комп'ютерного зору це правильно знайдений баланс між параметрами відео (розширення, формат, кількість кадрів на секунду) і технічними можливостями системи обробляти такі потоки інформації в реальному часі, а отже тема пошуку оптимального методу обробки потокового відео актуальна як ніколи раніше.

Тема також є дуже актуальною конкретно для платформи Node.js [2], оскільки дана платформа поєднує в собі продуктивність, модульність і є доволі ефективною для створення мікросервісних [3-4] систем. А оскільки у разі застосування комп'ютерного зору справедливо говорити про не одне а декілька джерел відео то постають задача уніфікації кадрів, їх обробки а також їх синхронізації і передачі безпосередньо до системи комп'ютерного зору. Даний принцип органічно втілюється за допомогою мікросервісної

архітектури, а Node.js добре проявляє себе в даній сфері, тому є доволі популярною, наприклад, в сфері IoT [5], яка за своїм визначенням є близькою до описаної вище системи.

Мета роботи – дослідження методів обробки потокового відео для платформи Node.js. Встановлена мета обумовлює наступні завдання:

- Формулювання вимог до процесу обробки і до результату
- Визначення найпопулярніших підходів а також параметрів відео
- Написання програмного коду для кожного з підходів
- Автоматизація роботи з відео і збору метрик
- Тестування, аналіз і порівняння результатів
- Створення порівняльної характеристики і пошук підходу який найкраще задовольняє вимоги

Об’єктом дослідження є методи обробки потокового відео на базі платформи Node.js, їх ефективність в різних умовах і при виконанні різних задач.

Предметом дослідження є пошук оптимальних підходів до обробки потокового відео на базі платформи Node.js при використанні різних роздільних здатностей, кодеків і для виконання різноманітних задач.

Наукова новизна дослідження полягає у створенні порівняльної характеристики бібліотек для обробки потокового відео при виконанні ряду задач з застосуванням ряду кодеків [6]. Дана інформація може бути знайдена у вигляді теорії, однак існуючих досліджень на цю тему недостатньо щоб впевнено казати про перевагу одного кодека над іншим, або однієї бібліотеки над іншою, окрім очевидних ситуацій, що будуть описані пізніше

Практичне значення роботи визначається створенням порівняльної характеристики, що чітко висвітлює переваги і недоліки існуючих технологій. Даним дослідженням слід керуватись у разі розробки систем з застосуванням комп’ютерного зору з великою кількістю камер, адже потреба в оптимізації даних систем зростає з кількістю даних, які потребують синхронної обробки.

В роботі було застосовано існуючі технології кодування відео (кодеки) і роботи з відео (бібліотеки для обробки відео). Був проведений збір таких метрик як навантаження на ЦП і використання оперативної пам'яті, а також візуальний аналіз отриманих відео на предмет зниження FPS, передчасного завершення відео, графічних артефактів [7] і інших свідчень того, що поставлена задача не може бути повною мірою виконана.

РОЗДІЛ 1. ПОСТАНОВКА ЗАДАЧІ ТА АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ РОБОТИ

1.1 Огляд процесу обробки потокового відео

Для того щоб визначити чітку задачу, необхідно спочатку визначити ключові аспекти предметної області. Тому варто розібрати обробку потокового відео на складові.

Процес обробки потокового відео не є чимось новим для інформаційних технологій. Потреба в обробці відео була очевидною одразу після появу ідеї цифрового відео, а потокове відео лише задає вимогу до обробки, а саме те, що система має бути достатньо ефективною, щоб не викликати затримок.

Для початку треба проаналізувати даний процес. Почати варто з обробки статичних зображень [8], адже історично кодеки починали з використання масиву зображень. Найбільший вплив на процес обробки, особливо в контексті потокового відео мають такі параметри як роздільна здатність і формат зображення.

Роздільна здатність, тобто ширина і висота зображення, що вимірюється в пікселях. Вище розширення означає більше роздільну здатність, а значить кращу якість і чіткість зображення, але й таке зображення однозначно займатиме більший об'єм пам'яті, і потребуватиме більше часу для обробки.

Формат зображення. Доволі часто є потреба зміни одного формату на інший. Такий процес називається транскодуванням [9-10]. Скоріше за все, сучасний кодек не використовуватиме вже існуючий формати зображень для кадрів відео, але винятки є.

Отже найбільший вплив на процес обробки зображень має їх формат і розширення. Далі варто переходити до відео. Raw video, тобто «сире», не кодоване відео є потоком зображень, тобто кадрів, відповідно внесення змін до відео включає в себе описану вище роботу з зображеннями, адже розширення відео є розширенням кожного з його кадрів, а формат диктує певний формат його кадрів. Нині raw video використовується у разі потреби в максимальній можливій якості, наприклад для професійної зйомки. У всіх

інших ситуаціях використовується кодування відео для спрощення його обробки і зменшення необхідного обсягу пам'яті. Опишемо основні підходи до кодування відео, [11-12] а саме inter-frame compression і intra-frame compression:

Intra-frame compression або intra-frame coding, тобто інтракадрове стиснення/кодування. Тобто принцип стиснення всього відео полягає в стисненні кожного з кадрів. Одним з кодеків, який використовує даний підхід є MJPEG, тобто Motion-JPEG. Його назва пояснює принцип роботи, відео з кодуванням MJPEG фактично є масивом зображень у поширеному форматі JPEG [13], який відомий своїм алгоритмом стиснення з втратами. Але варто зазначити що даний підхід дозволяє використання як алгоритмів стиснення зображень з втратами, так і без втрат, якщо в цьому є потреба [14].

Inter-frame compression [15], або просто inter frame, тобто інтеркадр, інтеркадрове стиснення. Даний підхід не вносить змін до кадрів. Фактично, він відходить від ідеї використання кадрів, в звичному їх розумінні. Зберігається різниця між кадрами. Такий принцип дозволяє суттєво знизити обсяг пам'яті, яку займає відео, але робота з таким відео складніша і вимагає більше часу на обробку. Кодек H.264 є класичним прикладом застосування inter-frame compression, наприклад такі гіганти як YouTube і Twitch активно застосовують його. Спільною рисою з intra-frame compression рисою є можливість використання як алгоритмів з втратами, так і без. Суть роботи даного стиснення полягає в наступному. Існує три види кадрів, I, P та B. I-frame це опорний кадр, зміни відносно якого і будуть збережені у наступних кадрах. I-frame створюється раз на певний період часу. P-frame це проміжний опорний кадр, він зберігає різницю між собою і I-frame, або попереднім P-frame. B-frame це звичайний кадр, і для більшого стиснення він зберігає в собі зміни як відносно попереднього опорного кадру, так і відносно наступного. Це дозволяє забезпечити дуже ефективне стиснення. Також слід зазначити що сучасні кодеки часто модифікують даний принцип і дозволяють таким кадрам

працювати не лише з I та P кадрами, як з опорними, так і з іншими B кадрами. Принцип роботи inter-frame compression продемонстровано на рис. 1.1.

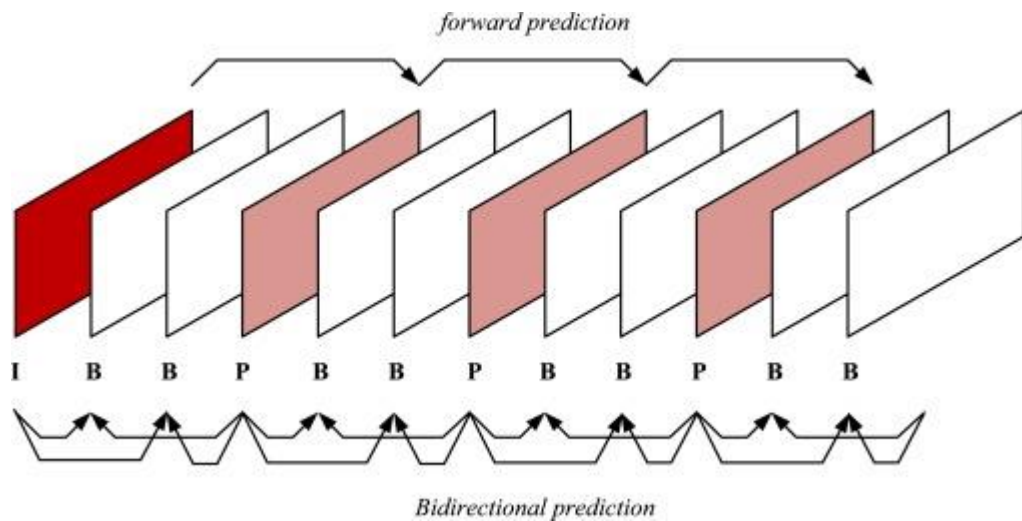


Рис. 1.1. Принцип роботи inter-frame compression

Також на обробку відео впливає FPS, тобто кількість кадрів на секунду. Логічно що при однаковому хронометражі відео відео з меншим FPS буде оброблятися швидше. Також цілком закономірним є те, що сам процес зниження FPS потребує меншої кількості ресурсів ніж звичайна зміна кодеку, тобто транскодування, адже при меншому fps кількість кадрів, які слід зашифрувати пропорційно зменшується.

Різниця між обробкою звичайного і потокового відео полягає в тому, що кожен відеопакет (кадр або пакет з даними про різницю між кадрами) має бути оброблений раніше ніж буде отримано наступний пакет. Відповідно, при FPS=5 задача полягає в обробці пакету менше ніж за 200 мілісекунд, і чим більший цей показник тим менше часу у системи на обробку. В парі «джерело відео»-«обробник відео» кількість кадрів буде або обмежена можливостями джерела, або можливостями обробника, адже у разі обробки довшої ніж дозволено наступні пакети або буде проігноровано до тих пір поки робота над поточним пакетом не буде завершена, або буде з'являтися затримка в силу того що кадри обробляються довше ніж час між кадрами. Навіть мінімальна затримка є неприпустимою, адже вона з часом буде накопичуватись, створюючи значні неприємності для роботи системи в цілому.

1.2. Постановка задачі

Головною метою кваліфікаційної роботи є дослідження методів обробки потокового відео на базі платформи Node.js, аналіз і порівняння їх ефективності для роботи з тими чи іншими кодеками. Виконане дослідження дозволить отримати практичні результати, що покажуть різницю в продуктивності між технологіями, а також визначити їх переваги і недоліки для виконання тих чи інших задач.

Нині одна з найбільших сфер застосування потокового відео це системи комп'ютерного зору [16]. Крім цього, дана сфера має серйозні вимоги щодо оптимізації системи і споживання мінімальної можливої кількості ресурсів в силу своєї архітектури. Часто можна зустріти системи комп'ютерного зору, що потребують значної кількості камер, і працюють за рахунок синхронного аналізу даних з кожної з них [17]. В ідеальних умовах дані камери мають надсилати для подальшої обробки відео, яке вже відповідає всім вимогам щодо кодеку, кількості кадрів на секунду, роздільної здатності, а також налаштувань зображення, адже залежно від позиції камера може вимагати індивідуальних налаштувань. Все це досить складно досягнути, особливо якщо розробник-не велика корпорація з прямим доступом до виробництва необхідних комплектуючих. Крім того, така ситуація створює доволі значне навантаження на сервер, адже тепер крім основних задач він мусить безпосередньо контролювати зображення з камер.

Саме тому часто вигіднішою є продемонстрована нижче схема. Камери групують і додають обробник відео до кожної з груп. Кількість камер в групі диктується лише можливостями обробника. Задача обробника полягає в обробці відео з камер до того стану, в якому воно максимально зручне для подальшої роботи вже на сервері.

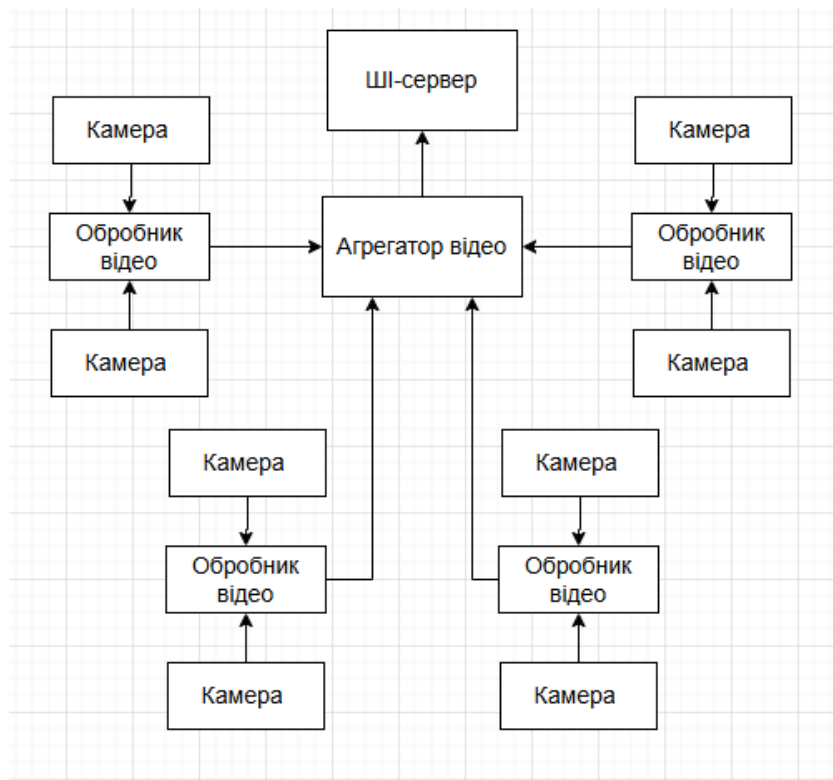


Рис. 1.2. Принцип побудови системи з застосуванням комп'ютерного зору

Кожен з елементів системи виконує певну функцію.

Зображення з камер в результаті потрапляє до ШІ-серверу. Змоделюємо операції, які відбуваються з зображенням в процесі передачі даних.

Зображення береться з камер, їх кількість обмежується лише потребами проекту. Одну або декілька камер можна під'єднати напряму до ШІ-серверу, але коли їх більше – з'являється потреба в проміжних ланках.

Зображення з камер поступає до обробників відео для перетворення в відео, яке є найбільш зручним в сприйнятті для ШІ. Наприклад, може бути змінена контрастність, роздільна здатність, кадрова частота, відеокодек.

Обробник відео передає дані до агрегатора відео, його задачею є синхронізація зображення з багатьох камер. Також в деяких системах він може відсіювати зайві камери, зображення з яких не несе корисної інформації.

Синхронні дані поступають на ШІ-сервер, який робить висновки згідно з поставленою йому задачею.

Система також може мати інші джерела інформації, такі як датчики ваги, руху, лідари і т.д. залежно від поставленої мети.

Задачі, які може виконувати обробник:

Базовою задачею є транскодування, тобто зміна кодека, який використовується для відео. Майже будь-яка маніпуляція з відео вимагає транскодування, адже зміни мають вносити в гав, тобто сире, не закодоване відео. Тобто відео декодується, вносяться необхідні зміни, після чого відео знову кодується з застосуванням потрібного кодеку.

Прикладом комплекснішої операції є зміна роздільної здатності. Відбувається транскодування, але перед кодуванням в цільовий кодек змінюється роздільна здатність, що може спростити роботу з відео в подальшому, особливо якщо відбувається зміна в менший бік, адже кадри стають меншими і кодувати тепер треба менший обсяг даних.

Зміна FPS [19] працює аналогічно з попередньою задачею, але після декодування відбувається кодування з частковим пропуском кадрів. Для людського ока таке відео буде не настільки комфортним, але досить часто системи комп'ютерного зору не потребують 24 або 30 кадрів на секунду, що є стандартом для компактних бюджетних камер, і зниження кадрової частоти може призвести до зменшення навантаження на кожен з елементів системи.

Іноді є потреба в зміні параметрів відео, наприклад балансу білого чи контрасту. Потреба в додаванні на відео додаткових елементів малоїмовірна, але все ж виключати її не варто.

Відповідно, є потреба в дослідженні таких процесів як транскодування, зміна роздільної здатності, зміна FPS, і внесення змін безпосередньо у відеоряд. Було обрано саме додавання додаткового елементу, оскільки даний процес є складнішим і його дослідження буде більш ілюстративним.

Також, слід визначитись з кодеками що візьмуть участь в дослідженні. Найпопулярнішим кодеком для IP-камер є H.264 [20], отже для вхідного відео було обрано саме його. Але серед USB-камер найбільш поширеним є кодек MJPEG а значить тестування варто провести з застосуванням обох кодеків, у

разі H.264 варто виконати емуляцію IP-камери, а для кодека MJPEG можна застосувати вбудовану веб-камеру ноутбука, оскільки конструктивно вона є USB-камерою. У якості цільових кодеків було обрано найпопулярніші на даний момент кодеки.

Першим з популярних кодеків, що використовують inter-frame compression став вже класичний кодек H.264 або AVC, що вже довгий час утримує першу позицію за популярністю. Використовує inter-frame compression, за рахунок цього може запропонувати ефективне стиснення без серйозних втрат якості. Розбиває кадр на макроблоки і працює за рахунок їх порівняння [21]. Підтримується багатьма контейнерами, наприклад flv, mp4, avi, mkv і інші

Пізніше була створена покращена версія H.264, під назвою H.265 або HEVC для того щоб прийти йому на заміну. Вимагає потужнішої системи для коректної роботи, однак має забезпечувати значно ефективніше стиснення. Працює за подібним принципом, однак використовує не однакові макроблоки 16x16 пікселів, а розбиває зображення на області ситуативно, за рахунок чого стискає краще. Однак, такий алгоритм є складнішим [22]. Підтримується меншою кількістю контейнерів, однак обрати все ще є з чого, наприклад mp4, mkv, avi і деякі інші менш поширені контейнери.

Приблизно одночасно з H.265 з'явився кодек VP9. Це була спроба конкурувати з H.265, згідно з заявами розробників має здійснювати ефективніше стиснення і з точки зору декодування бути оптимізованим для роботи на малопотужних пристроях [23]. Проте, даний кодек не отримав настільки широкої популярності. Підтримується контейнером webm

Найсучаснішим кодеком в даному переліку однозначно є AV1, його розробники ставили за мету перевершити H.265 і VP9. В основному розроблявся для інтернет-транслявання відео, заявляється про високу ефективність стиснення, однак в силу своєї новизни підтримується не всіма бібліотеками і далеко не всією технікою [24] Аналогічно до VP9 підтримується контейнером webm. Не братиме участі в дослідженні через те, що апаратна

підтримка даного кодеку доступна на сучасніших процесорах ніж є в наявності.

Далі слід розглянути кодеки з intra-frame compression. І почати має сенс з кодеку, який відомий високою якістю зображення, адже це не надто характерно для кодеків з intra-frame compression. Це кодек ProRes що був розроблений компанією Apple. За мету ставилась максимальна якість і простота в декодуванні, тому за своїми параметрами даний кодек нагадує raw video. За стисненням в значно поступається іншим, але пропонує мінімум втрат, або й повну їх відсутність [25]. Підтримується контейнерами mov і mp4

Доволі поширеним варіантом кодування відео з втратами є MPEG 4 Part 2. Досить старий кодек, який подекуди досі можна зустріти. Використовує intra-frame compression, має певний рівень втрат. Доволі легко обробляється, але не надає такого ж ефективного стиснення як деякі інші кодеки [26]. Контейнером для нього може служити mp4

Дуже подібним до попереднього є кодек MJPEG, що фактично є масивом JPEG-зображень, з усіма наслідками такого підходу. Втрати, навіть при низькому їх рівні доволі помітні, стиснення теж далеко не найкраще [27]. Проте цей кодек досить широко розповсюджений, найчастіше зустрічається при спробі читати відео з USB-камери. Підтримується контейнерами mp4 і avi

Також серед відомих кодеків часто можна зустріти такий кодек як Theora. Раніше доволі амбіційний кодек, який нині все ще використовується, але велика частина бібліотек відмовились від його підтримки, адже він застарів і не витримує конкуренції. Незважаючи на певне обмежене використання, в дослідженні він участі не братиме. Підтримується доволі рідкісним на даний час контейнером ogg

Отже, варто порівняти методи обробки потокового відео на прикладі перелічених вище операцій для кожного з кодеків, окрім Theora через його застарілість і AV1 через повну відсутність його підтримки системою, на якій буде виконано тестування. Відповідно, 6 різних кодеків.

Також доцільним є дослідити вплив роздільної здатності на операції.

З метою спрощення процесу аналізу даних, замість подальшої передачі відео у вигляді потоку, як це було б у справжньому проекті, доцільно буде зберігати результат у файл. Таким чином процес аналізу не заважатиме процесу збору метрик, а сам аналіз стане значно простішим у реалізації.

1.3. Обґрунтування мови програмування для якої буде здійснюватися дослідження

Дослідження проводиться з метою подальшого застосування отриманих результатів у проектах, які використовують комп'ютерний зір, а значить технології треба підбирати відповідні до специфіки подібних проектів. Архітектура подібних проектів була описана раніше

Для створення обробника відео доцільно використати платформу Node.js, адже вона пропонує баланс між зручністю розробки і швидкістю роботи, а також має значний потенціал в плані асинхронності [28], що однозначно стане в нагоді у разі великої кількості підключених камер. До того ж, Node.js не займає значного обсягу пам'яті і не потребує значних ресурсів системи, на відміну від деяких конкурентів на ринку. А це значить що застосування цієї платформи можливе і на слабких системах, наприклад на одноплатних комп'ютерах, чи інших компактних комп'ютерах, адже використовувати повноцінний комп'ютер для лише декількох камер часто неефективно.

Також можна відкинути дослідження обробки потокового відео для мов C/C++. Вони дуже ефективні в даній сфері за умови правильного використання, але є складнішими і менш придатними для написання мікросервісних систем. Але фактично дане дослідження зачіпатиме їх, адже бібліотеки, що будуть досліджені написані саме з застосуванням цих мов, так само як і платформа Node.js.

Може здатись що гарна ідея користуватись мовою програмування Python, оскільки III імовірніше за все буде написаний з застосуванням саме цієї мови, але це рішення є доволі спірним. Все ж в плані швидкодії і асинхронності Node.js показує кращі результати [29], в той час як Python має

більшу кількість модулів для використання в цій і не тільки сферах. Але швидкодія це все ж пріоритет, тому варто зупинитись на Node.js

Node.js дає можливість використовувати дві мови, JavaScript і TypeScript [30]. Фактично, TypeScript є покращеним JavaScript зі статичною типізацією, що робить розробку значно приємнішою. Але при запуску програми все одно запускається створений автоматично JavaScript-код, тобто відмінності в роботі майже відсутні. Тому TypeScript є однозначно кращим вибором.

1.4. Аналіз існуючих рішень в сфері обробки потокового відео:

Ffmpeg [31] – це потужний мультимедійний інструмент із відкритим вихідним кодом, який дозволяє записувати, конвертувати, транслювати та обробляти відео та аудіо. Його назва походить від **Fast Forward MPEG**, що вказує на його початковий фокус на обробці MPEG-файлів.

Основні функції Ffmpeg [32] це транскодування, зміна формату, розрізання і об'єднання відео або аудіо, додавання фільтрів, зміна параметрів відео, зміна роздільної здатності, частоти кадрів, бітрейту, тобто якості, отримання технічної інформації, такої як кодек, кадрова частота, роздільна здатність і іншої інформації про файли і потоки.

Ffmpeg підтримує прийом і виведення відео за допомогою файлів, а також HTTP, RTSP, RTP, HLS, MPEG-DASH і інших потоків.

Широка підтримка форматів, навіть дуже рідкісних, висока продуктивність, гнучкість завдяки наявним параметрам і фільтрам є дуже значними перевагами бібліотеки Ffmpeg.

Однак Ffmpeg є досить складним інструментом. Однією з причин такої складності є використання інтерфейсу командного рядка, що разом з великою кількістю параметрів робить читання команди доволі складною задачею.

Fluent-ffmpeg – це JavaScript-бібліотека для роботи з FFmpeg у середовищі Node.js. Вона надає зручний API для керування функціональністю FFmpeg, спрощуючи виконання складних мультимедійних операцій.

Функції і переваги лишаються тими ж, адже це лише обгортка для ffmpeg, створена щоб зробити роботу з ним легшою і приємнішою

Однак, подібні модифікації можуть призвести до появи таких недоліків як втрата продуктивності, потенційне відставання можливостей даного модуля від оригінального Ffmpeg, а також гірша гнучкість, адже замість параметрів командного рядка було додано методи які приймають значення параметрів.

GStreamer [33] – це мультимедійний фреймворк із відкритим кодом, який використовується для створення, обробки, передачі та відтворення аудіо-та відеопотоків. Він забезпечує високу гнучкість і потужність, дозволяючи розробникам легко налаштовувати та комбінувати компоненти для роботи з мультимедіа.

Основні функції GStreamer це транскодування, зміна формату, додання фільтрів, розрізання аудіо та відео, зміна частоти кадрів, роздільної здатності, бітрейту.

Gstreamer дозволяє прийом і виведення відео за допомогою файлів, а також великої кількості різних потоків, таких як RTSP, RTP, HTTP, HLS

Модульна архітектура і конвеєрний принцип роботи є однозначними перевагами бібліотеки GStreamer. Також він має підтримку деяких плагінів, що входять у склад FFmpeg.

Однозначним недоліком є складність. Команда для запуску Gstreamer є ще більше складною і перевантаженою ніж для FFmpeg. Крім того, модульна архітектура може бути і недоліком, адже процес додання нових плагінів є складним і може нашкодити вже існуючим. Також Gstreamer займає значно більше місця на диску у порівнянні з FFmpeg.

Висновки до першого розділу

Даний розділ охоплює теоретичну інформацію про інструменти і підходи для обробки потокового відео, аналіз існуючих бібліотек, модулів і фреймворків які здатні виконувати ці функції і коротке порівняння популярних відео кодеків.

В результаті проведеного дослідження було вирішено використовувати мову програмування TypeScript для створення тестувального середовища.

Згідно результатів отриманих в розділі 1 було вирішено розробити тестове середовище, порівняти роботу різних інструментів для різних задач, параметрів відео і кодеків, після чого сформувавши порівняльну характеристику і знайти оптимальні підходи до обробки потокового відео

РОЗДІЛ 2. ПРАКТИЧНА РЕАЛІЗАЦІЯ ОБРОБКИ ВІДЕО

2.1. Реалізація пробного транскодування відео з застосуванням обраних інструментів

Для початку слід встановити необхідні бібліотеки. Ffmpeg встановлюється на Ubuntu за допомогою наступної команди:

```
apt install ffmpeg
```

Gstreamer встановити дещо складніше. Треба запустити наступну команду:

```
apt-get install libgstreamer1.0-dev libgstreamer-plugins-base1.0-dev libgstreamer-plugins-bad1.0-dev gstreamer1.0-plugins-base gstreamer1.0-plugins-good gstreamer1.0-plugins-bad gstreamer1.0-plugins-ugly gstreamer1.0-libav gstreamer1.0-tools gstreamer1.0-x gstreamer1.0-alsa gstreamer1.0-gl gstreamer1.0-gtk3 gstreamer1.0-qt5 gstreamer1.0-pulseaudio
```

Крім цього, Gstreamer може вимагати встановлення додаткових плагінів. Для цього доведеться завантажувати ці плагіни, компілювати власноруч. Такі маніпуляції, якщо виконані не надто охайно, можуть призвести до відмов при роботі з ffmpeg, адже частина бібліотек для них є спільними, відповідно ручне налаштування може зачіпати обидва інструменти

Ffmpeg і Gstreamer є інструментами, що використовують інтерфейс командного рядка, а отже для початку слід скласти пробну команду для транскодування і розібрати її на складові, після чого перенести дану команду на мову TS.

```
ffmpeg -i rtsp://localhost:8554/stream -c:v libx265 -f flv output.flv
```

Варто розібрати кожен з параметрів

-i визначає джерело відео потоку

-c:v задає відеокодек, у даному випадку H.265. На прикладі попереднього і цього параметру можна помітити що ffmpeg має власні позначення для різних кодеків і вони можуть не співпадати з їх назвами [35].

-f формат вихідного файлу

output.flv це файл для виведення результату. При застосуванні в реальних умовах буде замінено на URL для подальшої трансляції

Відповідно реалізація такої команди для TypeScript виглядатиме так:

```
transcode(outputCodec: VideoCodec, resolution: Resolution): void {
    const { codec, format } =
SimpleFfmpeg.getInitialVideoConfigForFfmpeg(outputCodec);
    const filename = getFileName(Task.TRANSCODE, this.getName(), outputCodec,
resolution);
    this.ffmpegProcess = spawn('ffmpeg', [
        '-i', constants.INPUT_STREAM,
        '-c:v', codec,
        '-f', format,
        `${constants.OUTPUT_FOLDER}/${filename}.${format}`,
    ]);
    this.addListeners();
}
```

Подібна структура буде використовуватись і в подальшому, отже слід одразу її розібрати.

Метод `getInitialVideoConfigForFfmpeg()` відповідає за перетворення позначення кодеку у звичному для нас форматі, наприклад H.264, VP9 на формат, зрозумілий для бібліотеки Ffmpeg, тобто `libx264` і `libvpx-vp9` відповідно. А також даний метод визначає контейнер, який зможе працювати з даним кодеком. Для H.264 було обрано `flv`, а для VP9 це `webm`. Сам метод має такий вигляд:

```
static getInitialVideoConfigForFfmpeg(codec: string): { codec: string,
format: string } {
    switch (codec) {
        case VideoCodec.H264:
            return { codec: 'libx264', format: 'flv' };
        case VideoCodec.H265:
            return { codec: 'libx265', format: 'mp4' };
        case VideoCodec.MJPEG:
            return { codec: 'mjpeg', format: 'avi' };
        case VideoCodec.MPEG4_PART_2:
            return { codec: 'mpeg4', format: 'mp4' };
        case VideoCodec.ProRes:
            return { codec: 'prores_ks', format: 'mov' };
        case VideoCodec.VP9:
            return { codec: 'libvpx-vp9', format: 'webm' };
        default:
            return { codec: 'libx264', format: 'flv' };
    }
}
```

Метод `getFileName()` формує ім'я файлу у форматі ОПЕРАЦІЯ-ІНСТРУМЕНТ-КОДЕК-РОЗМІР. Наприклад `TRANSCODE-SimpleFfmpeg-H.265-1280x720`

```
export function getFileName(task: string, tool: string, codec?: VideoCodec,
resolution?: Resolution): string {
    return `${task}-${tool}-${codec ? codec : VideoCodec.H265}-${resolution ?
resolution : Resolution.HD}`;
}
```

Метод `addListener()` підписується на події, що будуть публікуватись в процесі роботи бібліотеки. Основна причина це доступ до логів, що значно спрощує розробку.

```
private addListeners(): void {
  if (this.ffmpegProcess) {
    this.ffmpegProcess.stdout.on('data', (data) => {
      console.log(`FFmpeg output: ${data}`);
    });

    this.ffmpegProcess.stderr.on('data', (data) => {
      console.error(`FFmpeg error: ${data}`);
    });

    this.ffmpegProcess.on('close', (code) => {
      console.log(`FFmpeg process closed with code: ${code}`);
    });
  }
}
```

Далі слід аналогічно реалізувати транскодування для `fluent-ffmpeg`. Дана бібліотека є лише надбудовою над `ffmpeg`, що спрощує роботу але не змінює суті. У даному прикладі.

```
transcode(outputCodec: VideoCodec, resolution: Resolution): void {
  const { codec, format } =
    SimpleFfmpeg.getInitialVideoConfigForFfmpeg(outputCodec);
  const filename = getFileName(Task.TRANSCODE, this.getName(), outputCodec,
    resolution);
  this.ffmpeg = ffmpeg(constants.INPUT_STREAM)
    .output(`${constants.OUTPUT_FOLDER}/${filename}.${format}`)
    .videoCodec(codec)
    .format(format)
    .on('start', (commandLine) => {
      console.log('FFmpeg process started:', commandLine);
    })
    .on('end', () => {
      console.log('FFmpeg process finished.');
```

Дана операція для `Gstreamer` буде виглядати наступним чином:

```
gst-launch-1.0 rtspsrc location=rtsp://localhost:8554/stream
protocols=4 latency=500 ! decodebin ! videoconvert ! x265enc ! h265parse
! matroskamux -e ! filesink location=output.mkv
```

Розберемо кожен з параметрів:

`rtspsrc` задає джерело відео, а також основні налаштування щодо джерела, такі як максимальна затримка, і протоколи що будуть використовуватись для отримання даних.

decodebin означає те що gstreamer має декодувати відео з джерела, тобто отримати raw video. Gstreamer може визначити вхідний кодек автоматично, але за потреби замість decodebin можна вказати декодер, який буде використовуватись для декодування вхідного відео.

videoconvert означає потребу в кодуванні відео.

x265enc це енкодер, тобто плагін що відповідає за кодування відео у кодек H.265.

h265parse це парсер, який перетворює потік H.265 на формат, що підходить для подальшої передачі або зберігання. Він додає кадрам заголовки і готує потік до мультиплексування. Суто теоретично, вказувати його не обов'язково і іноді конвеєр працюватиме і без нього, але гарантувати це неможливо, результат залежить від кодека і самого конвеєра.

matroskamux відповідає за мультиплексування у формат mkv. Параметр -e означає гарантію коректного завершення файлу. При перериванні роботи конвеєра, отриманий в результаті файл може виявитись пошкодженим і відтворити його буде неможливо.

filesink визначає файл, в який буде збережено результат роботи конвеєру.

Відповідно, код який виконує описану вище команду, виглядатиме так:

```
transcode(outputCodec: VideoCodec, resolution: Resolution): void {
  const gstCommand = 'gst-launch-1.0';
  const { codec, format, parser, fileExtension } =
Gstreamer.getInitialVideoConfigForGstreamer(outputCodec);
  const filename = getFileName(Task.TRANSCODE, this.getName(), outputCodec,
resolution);
  const inputLine = this.getInputLine();
  const parserLine = parser ? [parser, '!!'] : '';
  const gstArgs = [
    ...inputLine, '!!',
    'decodebin', '!!',
    'videoconvert', '!!',
    codec, '!!',
    ...parserLine,
    format, '-e', '!!',
    'filesink',
    `location=${constants.OUTPUT_FOLDER}/${filename}.${fileExtension}`
  ];
  this.gstreamerProcess = spawn(gstCommand, gstArgs, { env: {
...process.env, GST_DEBUG: '3' } });
  this.addListeners();
}
```

Метод `getInitialVideoConfigForGstreamer()` виконує ту ж функцію що аналогічний метод для `Ffmpeg`, функція `getFileName()` була продемонстрована раніше.

Специфічним є метод `getInputLine()`, він призначений для формування аргумента, що стосується джерела потоку. Це необхідно для легкого переходу з RTSP-трансляції на використання USB-камери.

```
getInputLine(): string[] {
    if (constants.INPUT_CODEC.toString() === VideoCodec.MJPEG.toString()) {
        return ['v4l2src', `device=${constants.INPUT_STREAM}`];
    }

    return ['rtspsrc', `location=${constants.INPUT_STREAM}`, 'protocols=4',
'latency=500'];
}
```

Різниця полягає в тому що для використання USB-камери слід використовувати `v4l2src` замість `rtspsrc`.

Також можна помітити опціональне використання парсера. Для пояснення варто навести лістинг методу `getInitialVideoConfigForGstreamer()`

```
static getInitialVideoConfigForGstreamer(codec: string): { codec: string,
format: string, parser?: string, fileExtension: string } {
    switch (codec) {
        case VideoCodec.H264:
            return { codec: 'x264enc', format: 'flvmux', parser: 'h264parse',
fileExtension: 'flv' };
        case VideoCodec.H265:
            return { codec: 'x265enc', format: 'matroskamux', parser:
'h265parse', fileExtension: 'mkv' };
        case VideoCodec.MJPEG:
            return { codec: 'jpegenc', format: 'avimux', parser: 'jpegparse',
fileExtension: 'avi' };
        case VideoCodec.MPEG4_PART_2:
            return { codec: 'avenc_mpeg4', format: 'qtmux', parser:
'mpeg4videoparse', fileExtension: 'mp4' };
        case VideoCodec.ProRes:
            return { codec: 'avenc_prores', format: 'qtmux', fileExtension:
'mov' };
        case VideoCodec.VP9:
            return { codec: 'vp9enc', format: 'webmmux', parser: 'vp9parse',
fileExtension: 'webm' };
        default:
            return { codec: 'x264enc', format: 'flvmux', parser: 'h264parse',
fileExtension: 'flv' };
    }
}
```

З показаного вище коду помітно що окрім вже знайомих кодека і типу контейнера метод повертає також тип мультимплексування, зачасту він відповідає типу контейнера, а крім цього-парсер. Суть у тому що кодек `ProRes`

не вимагає використання парсера, відповідно параметр парсера передавати не слід.

В результаті створення перших, тестових команд (у випадку Ffmpeg) і конвеєра для Gstreamer було продемонстровано основи роботи з даними технологіями. Можна чітко сказати що Gstreamer складніше у налаштуванні, але за рахунок цього здатний виконати більш точне і тонке налаштування під специфічні потреби. В той час як Ffmpeg і FluentFfmpeg є простішими і автоматизують частину процесів, які мають бути виконані.

Процес транскодування ляже в основу подальших операцій. Їх реалізація буде продемонстрована пізніше.

2.2. Формування вимог і проектування тестового середовища

Тестове середовище має в автоматичному режимі запускати кожну з операцій для кожного з відеокодеків з застосуванням кожного з інструментів. Глобальні параметри, на кшталт джерела відео, папки для збереження результатів і т.д. мають бути винесені окремо з метою простішого керування

Середовище має також збирати метрики під час виконання операції. Має сенс автоматично збирати дані про використання ЦП і оперативної пам'яті. У нагоді може стати як масив даних протягом всього часу виконання задачі, так і середнє, максимальне і мінімальне значення. Ці дані мають бути записані у файл.

Такі метрики як кількість кадрів на секунду, розмір файлу і його тривалість легко збираються вручну, але вони теж можуть стати в нагоді. Їх можна отримати з файлу, в який і буде записане відео частково візуально, частково за допомогою вбудованих інструментів відео програвача. Крім того, не менш важливим показником, який варто відслідковувати є наявність графічних артефактів. Вони є симптомом проблем з обробкою відео, їх наявність може свідчити або про неправильний підбір інструментів, або про неможливість виконати задачу з наявними обчислювальними потужностями.

Оскільки кожен з інструментів має виконувати однакові функції, доцільно створити батьківський клас, який оголошуватиме можливі операції, а також метод для зупинки процесу. Процес має перериватись по таймеру для того щоб поставити всі інструменти і кодеки в рівні умови.

Отже основна функція має запускати кожен процес для кожного з кодеків використовуючи кожен з інструментів по черзі, зупиняти їх через визначений період часу (наприклад 25 секунд), давати невеликий проміжок часу для коректного завершення роботи процесу (5 секунд більш ніж достатньо), запускати наступний процес. Дана неперервність може бути досягнена за допомогою таймерів, оскільки завдяки чітким часовим рамкам виконання роботи можна підрахувати скільки часу займе виконання тієї чи іншої кількості задач.

2.3. Створення тестового середовища

В першу чергу слід описати набір відеокодеків, задач і роздільних здатностей, що будуть застосовані

```
export enum VideoCodec {
    H264 = 'H264',
    H265 = 'H265',
    MJPEG = 'MJPEG',
    ProRes = 'ProRes',
    MPEG4_PART_2 = 'MPEG4_PART_2',
    VP9 = 'VP9',
}

export enum Resolution {
    Low240p = '427x240',
    LowForH265 = '428x240',
    HalfHD = '640x360',
    SD = '854x480',
    HD = '1280x720',
}

export enum Task {
    TRANSCODE = 'TRANSCODE',
    RESIZE = 'RESIZE',
    SET_FPS = 'SET_FPS',
    WATERMARK = 'WATERMARK'
}
```

Також, варто описати основний функціонал класів, що відповідають за інструменти

```
export abstract class VideoModuleTester {
    abstract getName(): string;
    abstract transcode(outputCodec: VideoCodec, resolution: Resolution):
void;
    abstract resize(videoCodec: VideoCodec, outputResolution: Resolution):
void;
    abstract setFps(outputCodec: VideoCodec, fps: number): void;
    abstract addWatermark(videoCodec: VideoCodec): void;
    abstract killProcess(): void;
}
```

Після цього можна переходити безпосередньо до розробки інструментів. Транскодування вже було реалізовано раніше, тому можна переходити до інших операцій

Зміна роздільної здатності за допомогою Ffmpeg досягається доданням параметру `-vf` зі значенням цільової роздільної здатності. Іншою відмінністю є те, що при застосуванні кодека H.265 може використовуватись специфічна роздільна здатність. Це відбувається через те що відеокодек H.265 вимагає виключно парних показників роздільної здатності, а у разі зміни роздільної здатності на 457x240, що є половиною роздільної якості SD є потреба зробити ширину парною

```
resize(outputCodec: VideoCodec, outputResolution: Resolution): void {
    const { codec, format } =
SimpleFfmpeg.getInitialVideoConfigForFfmpeg(constants.INPUT_CODEC.toString())
;
    const { codec, format } =
SimpleFfmpeg.getInitialVideoConfigForFfmpeg(outputCodec);
    const filename = getFileName(Task.RESIZE, this.getName(), outputCodec,
outputResolution);
    let size = outputResolution;
    if (outputCodec === VideoCodec.H265 && size === Resolution.Low240p) {
        size = Resolution.LowForH265;
    }
    this.ffmpegProcess = spawn('ffmpeg', [
        '-i', constants.INPUT_STREAM,
        '-input_format', inputCodec,
        '-vf', `scale=${size}`,
        '-c:v', codec,
        '-f', format,
        `${constants.OUTPUT_FOLDER}/${filename}.${format}`,
    ]);
    this.addListeners();
}
```

Зміна роздільної здатності на FluentFfmpeg. Значна різниця відсутня, роздільна здатність відео передається параметром в функцію `size()`

```
resize(outputCodec: VideoCodec, outputResolution: Resolution): void {
    const { codec, format } =
SimpleFfmpeg.getInitialVideoConfigForFfmpeg(outputCodec);
    const filename = getFileName(Task.RESIZE, this.getName(), outputCodec,
outputResolution);
    this.ffmpeg = ffmpeg(constants.INPUT_STREAM)
```

```

        .output(`${constants.OUTPUT_FOLDER}/${filename}.${format}`)
        .videoCodec(codec)
        .size(outputResolution)
        .format(format)
        .on('start', (commandLine) => {
            console.log('FFmpeg process started:', commandLine);
        })
        .on('end', () => {
            console.log('FFmpeg process finished.');
```

Gstreamer. Основною відмінністю від попередніх інструментів є явне застосування змін до сирого відео, що бібліотека Ffmpeg робить автоматично. Через це необхідність транскодування для внесення змін може здаватись відсутньою, але насправді транскодування відбувається в будь-якому разі.

Вказаний вище недолік кодека H.265 оброблений і тут

```

resize(outputCodec: VideoCodec, outputResolution: Resolution): void {
    const gstCommand = 'gst-launch-1.0';
    const { codec, format, parser, fileExtension } =
Gstreamer.getInitialVideoConfigForGStreamer(outputCodec);
    const filename = getFileName(Task.RESIZE, this.getName(), outputCodec,
outputResolution);
    const parserLine = parser ? [parser, '!!'] : '';
    let size = outputResolution;
    if (outputCodec === VideoCodec.H265 && size === Resolution.Low240p) {
        size = Resolution.LowForH265;
    }
    const [width, height] = size.split('x');

    const gstArgs = [
        ...this.getInputLine(), '!!',
        'decodebin', '!!',
        'videoconvert', '!!',
        'videoscale', '!!',
        `video/x-raw, format=(string)I420, width=(int)${width},
height=(int)${height}`, '!!',
        'videoconvert', '!!',
        codec, '!!',
        ...parserLine,
        format, '-e', '!!',
        'filesink',
        `location=${constants.OUTPUT_FOLDER}/${filename}.${fileExtension}`
    ];

    this.gstreamerProcess = spawn(gstCommand, gstArgs, { env: {
...process.env, GST_DEBUG: '3' } });
    this.addListeners();
}
```

Зміна кількості кадрів на секунду досягається доданням параметру -r для Ffmpeg, доданням аналогічного параметру за допомогою методу

outputOptions() для FluentFfmpeg, і аналогічна до попередньої операція з сирим відео, але з метою зміни кадрів на секунду, за допомогою параметру framerate

Доцільніше звернути увагу на реалізацію додання водяного знаку до відео, адже дана задача комбінує в собі попередні. Для того щоб зробити задачу ще складнішою для обробки системою, на роль водяного знаку було обрано .gif файл, який також слід зациклити

Для стандартного Ffmpeg додання водяного знаку виглядає наступним чином:

```
addWatermark(videoCodec: VideoCodec): void {
    const { codec, format } =
SimpleFfmpeg.getInitialVideoConfigForFfmpeg(videoCodec);
    const filename = getFileName(Task.WATERMARK, this.getName(), videoCodec,
constants.INPUT_RESOLUTION.toString());
    const watermarkPath = path.join(process.cwd(), constants.WATERMARK_FILE);
    const ffmpegArgs = [
        '-i', constants.INPUT_STREAM,
        '-ignore_loop', '0',
        '-i', watermarkPath,
        '-filter_complex',
        '[1]format=rgba,colorchannelmixer=aa=0.8[logo];[0][logo]overlay=(W-
w)/2:(H-h)/2:format=auto,format=yuv420p',
        '-c:v', codec,
        '-f', format,
        `${constants.OUTPUT_FOLDER}/${filename}.mp4`,
    ];

    this.ffmpegProcess = spawn('ffmpeg', ffmpegArgs);
    this.addListeners();
}
```

-ignore-loop 0 означає те, що зациклення .gif файлу дозволяється

Використовується два параметра -i, для джерела відео і файлу водяного знаку

Після цього описане накладання одного зображення на інше, а саме прозорість водяного знаку 0.8, позиція по центру, формат пікселів yuv420p

Подальші параметри були описані раніше

FluentFfmpeg:

```
addWatermark(videoCodec: VideoCodec): void {
    const { codec, format } =
SimpleFfmpeg.getInitialVideoConfigForFfmpeg(videoCodec);
    const filename = getFileName(Task.WATERMARK, this.getName(), videoCodec,
constants.INPUT_RESOLUTION.toString());
    const watermarkPath = path.join(process.cwd(), constants.WATERMARK_FILE);
    this.ffmpeg = ffmpeg(constants.INPUT_STREAM)
        .input(watermarkPath)
        .inputOptions('-ignore_loop 0')
        .output(`${constants.OUTPUT_FOLDER}/${filename}.${format}`)
```

```

        .complexFilter([
' [1]format=rgba,colorchannelmixer=aa=0.8[logo];[0][logo]overlay=(W-w)/2:(H-h)/2:format=auto,format=yuv420p',
        ])
        .videoCodec(codec)
        .on('start', (commandLine) => {
            console.log('FFmpeg process started:', commandLine);
        })
        .on('end', () => {
            console.log('FFmpeg process finished.');
```

Параметри аналогічні, але передаються за допомогою методу `complexFilter` і `inputOptions`

Gstreamer:

```

addWatermark(videoCodec: VideoCodec): void {
    const gstCommand = 'gst-launch-1.0';
    const { codec, format, parser, fileExtension } =
Gstreamer.getInitialVideoConfigForGStreamer(videoCodec);
    const filename = getFileName(Task.WATERMARK, this.getName(), videoCodec);
    const parserLine = parser ? [parser, '!!'] : '';
    const watermarkPath = path.join(process.cwd(),
constants.LOOPED_WATERMARK_FILE);
    const gstArgs = [
        ...this.getInputLine(), '!',
        'decodebin', '!',
        'videoconvert', '!',
        'queue', '!',
        'compositor', 'name=comp',
        'sink_1:xpos=10', 'sink_1:ypos=10', '!',
        'videoconvert', '!',
        codec, '!',
        ...parserLine,
        format, '!',
        'filesink',
        `location=${constants.OUTPUT_FOLDER}/${filename}.${fileExtension}`,

        'filesrc', `location=${watermarkPath}`, '!',
        'decodebin', '!',
        'videoconvert', '!',
        'videorate', '!',
        'video/x-raw,framerate=24/1', '!',
        'queue', '!',
        'comp.sink_1'
    ];

    this.gstreamerProcess = spawn(gstCommand, gstArgs, { env: {
...process.env, GST_DEBUG: '4' } });
    this.addListeners();
}
```

Провести дану маніпуляцію за допомогою Gstreamer значно складніше. Крім того, Gstreamer не дозволяє зробити це настільки ж якісно, наскільки Ffmpeg

Параметр queue служить для синхронізації основного відео і водяного знаку.

Оголошується compositor, тобто елемент що відповідає за накладання одного зображення (кадр водяного знаку) на інший (кадр сирого відео)

Оголошується позиція майбутнього водяного знаку. Для різноманіття тут це верхній лівий кут

Після обробки основного відео оголошується обробка водяного знаку. Він теж декодується, задається частота кадрів на секунду, queue для синхронізації і водяний знак накладається за допомогою створеного раніше compositor.

Для того щоб запускати всі ці задачі в автоматичному режимі було створено основну функцію

```
function main(task: Task): void {
  const simpleFfmpeg = new SimpleFfmpeg();
  const fluentFfmpeg = new FluentFfmpeg();
  const gStreamer = new Gstreamer();
  const tools: VideoModuleTester[] = [simpleFfmpeg, fluentFfmpeg,
gStreamer];

  const processWatcher = new ProcessWatcher();

  let timeoutMultiplier = 0;
  for (const tool of tools) {
    for (const key in VideoCodec) {
      setTimeout(() => {
        logger.info(`Starting ${task} process with ${tool.getName()},
codec ${key}`);
        processWatcher.startWatching(tool.getName(), task,
constants.INPUT_RESOLUTION, key);
        switch (task) {
          case Task.TRANSCODE:
            tool.transcode(key, constants.INPUT_RESOLUTION);
            break;
          case Task.RESIZE:
            tool.resize(key, Resolution.Low240p);
            break;
          case Task.SET_FPS:
            tool.setFps(key, 15);
            break;
          case Task.WATERMARK:
            tool.addWatermark(key);
          }
        setTimeout(async () => {
```

```

        tool.killProcess();
        await processWatcher.stopWatching();
    }, 25 * 1000);
}, 30 * 1000 * (timeoutMultiplier));
timeoutMultiplier += 1;
    }
}
}

main(Task.TRANSCODE);
setTimeout(() => {
    main(Task.RESIZE);
}, 11 * 60 * 1000);
setTimeout(() => {
    main(Task.SET_FPS);
}, 22 * 60 * 1000);
setTimeout(() => {
    main(Task.WATERMARK);
}, 33 * 60 * 1000);

```

Було створено цикл, який призначає виконання конкретної задачі за таймером і зупиняє виконання задач після фіксованого періоду часу, таким чином виконуючи задачу кожним інструментом для кожного відеокодеку. Також за допомогою таймерів було призначено виконання різного роду задач. Час виконання задачі підраховується наступним чином: 30 секунд на кодек, 7 кодеків, 3 інструменти. $30 * 3 * 7 = 630$ секунд, тобто 10 хвилин і 30 секунд. Додамо 30 секунд про всяк випадок і таким чином можна сформувати таймер для наступної задачі.

Також було реалізовано автоматичний збір метрик щодо використання ресурсів процесора і оперативної пам'яті

```

export default class ProcessWatcher {
    private watcherTimeout: NodeJS.Timeout | undefined;
    private metrics: MetricPacket[] = [];
    private fileName: string | undefined;
    private codec: string | undefined;
    private tool: string | undefined;

    startWatching(tool: string, task: string, resolution: string, codec:
string): void {
        this.fileName = `${task}-${resolution}.json`;
        this.codec = codec;
        this.tool = tool;
        this.watcherTimeout = setInterval(async () => {
            const cpuUsage = +(await
si.currentLoad()).currentLoad.toFixed(1);
            const ramUsage = +(process.memoryUsage.rss() / (1024 *
1024)).toFixed(2);
            this.metrics.push({ cpuUsage, ramUsage });
        }, 300);
    }

    async stopWatching(): Promise<void> {
        clearTimeout(this.watcherTimeout);
    }
}

```

```

let sumCpu = 0;
let maxCpu = this.metrics[0].cpuUsage;
let minCpu = this.metrics[0].cpuUsage;
let sumRam = 0;
let maxRam = this.metrics[0].ramUsage;
let minRam = this.metrics[0].ramUsage;
this.metrics.forEach(({ cpuUsage, ramUsage }) => {
    sumCpu += cpuUsage;
    sumRam += ramUsage;
    if (cpuUsage > maxCpu) {
        maxCpu = cpuUsage;
    }
    if (ramUsage > maxRam) {
        maxRam = ramUsage;
    }
    if (cpuUsage < minCpu) {
        minCpu = cpuUsage;
    }
    if (ramUsage < minRam) {
        minRam = ramUsage;
    }
});
const avgCpu = sumCpu / this.metrics.length;
const avgRam = sumRam / this.metrics.length;
const overallMetrics = { codec: this.codec, tool: this.tool, minCpu,
maxCpu, avgCpu, minRam, maxRam, avgRam };
this.metrics = [];
this.fileName && await jsonfile.writeFile(path.join(process.cwd(),
constants.STATISTICS_FOLDER, this.fileName), overallMetrics, { flag: 'a',
EOL: ',' });
}
}

```

Даний функціонал буде дороблено для виконання ефективного тестування і застосовано для отримання відеофайлів і JSON-файлів з метриками і подальшого їх аналізу

Висновки до другого розділу

Протягом виконання другого розділу було проаналізовано принципи роботи досліджуваних інструментів, на основі даного аналізу-було реалізовано транскодування з подальшим збереженням у файл як базову операцію з потоковим відео

Було сформовано вимоги до майбутнього середовища тестування різних методів обробки потокового відео. Головними вимогами стали рівні умови для кожного з інструментів, автоматичний запуск і зупинка процесів роботи з відео, а також збір основних метрик.

Було реалізовано виконання транскодування, зміни роздільної здатності, зміни FPS і додавання водяного знаку для кожного з інструментів, додано

підтримку RTSP і USB джерел відео, а також реалізовано автоматичний запуск, збір метрик і зупинку виконання вказаних вище операцій.

РОЗДІЛ 3. ОТРИМАННЯ ДАНИХ І АНАЛІЗ ЕФЕКТИВНОСТІ ОБРОБКИ ПОТОКОВОГО ВІДЕО

3.1 Робота з тестовим середовищем

Було проведено тестування для трьох варіантів вхідного відео

Для кожного з них є індивідуальні налаштування обробки.

Таблиця 3.1

№	Джерело	Роздільна здатність	Цільова роздільна здатність	FPS	Цільовий FPS
1	RTSP	1280x720	640x360	25	15
2	RTSP	854x480	427x240	25	15
3	USB	1280x720	640x360	10	5

Операція додавання водяного знаку має однакові налаштування для кожного з варіантів вхідних даних.

Також, було вирішено використовувати 5 кадрів на секунду як цільове значення для USB-камери, оскільки початкове значення і так є значно нижчим ніж з RTSP-поток. Очевидно що намагатись підвищити FPS немає жодного сенсу.

Система на якій виконувалась робота:

Процесор: AMD Ryzen 2500U

Оперативна пам'ять: 16 ГБ DDR4

Відеокарта ролі не грає, оскільки обробка виконувалась на процесорі

Операційна система Ubuntu 22.04

3.2 Аналіз отриманих даних

Для аналізу даних слід виділити показники, на які варто звернути увагу

Високий відсоток використання ЦП може свідчити про те, що система працює на межі, і якщо проблем з зображенням ще не виникло, то ймовірність їх виникнення є дуже високою

Зниження FPS свідчить про те, що система не здатна виконати поставлену задачу в наявних часових рамках, тобто не встигає обробити кадр до того як буде отримано наступний

Також варто звернути увагу на якість відео. Вища якість є перевагою, але часто досягається більшим навантаженням на систему. Також, якість може

бути інструментом для зниження навантаження, адже за допомогою її регулювання можна спростити системі роботу

Розмір файлу свідчить про ступінь стиснення, яку здатен надати кодек. Аналогічно з якістю, за рахунок гіршого стиснення можна досягнути більшого розміру файлу але меншого навантаження на систему

У разі якщо відео триває відчутно менше ніж тривав процес обробки, тобто менше 20 секунд, оскільки процес займає 25 секунд, вимагає певний час для запуску а також може генерувати статичний кадр чи обрізати перші декілька секунд відео, можна говорити про те що процес не виконав поставлену задачу належним чином

Наявність графічних артефактів також є явним приводом вважати що система не здатна виконати поставлену задачу.

Першим інструментом, результати якого буде досліджено є SimpleFfmpeg.

Для першого варіанту вхідного відео, можна зробити такі висновки

Проблеми виникли при транскодуванні в такі відеокодеки як H265 і VP9

Відео H.265 мало проблеми з частотою кадрів на секунду, а також короткотривалі графічні артефакти (рис. 3.1), але по тривалості відповідало нормі.



Рис. 3.1. Графічні артефакти при застосуванні кодека H.265

Зауваження також є до відеокодека ProRes. У відео часто траплялись статичні кадри і графічні артефакти



Рис. 3.2. Графічні артефакти при застосуванні кодека ProRes

Аналогічні зауваження до відеокодека VP9, однак він став причиною ще більших проблем. Тривалість відео знизилась вдвічі, артефакти трапляються частіше і мають серйозніший характер (рис. 3.2).



Рис. 3.3. Графічні артефакти при застосуванні кодека VP9

Зниження роздільної здатності не мало суттєвих відмінностей від транскодування в плані результатів.

Зниження FPS призвело до подібних результатів. Тривалість відео з кодеком VP9 зросла до 15 секунд, частота появи графічних артефактів трохи знизилась.

Накладання анімованого водяного знаку є найскладнішою задачею серед поставлених, тому очікувано що три найскладніших кодека мали проблеми з її виконанням. А саме H.265, ProRes і VP9

Повноцінно виконати задачу зміг лише відеокодек MPEG 4 Part 2. Зауваження є лише до якості зображення, адже вона явно є найгіршою у порівнянні з іншими. Проте, порівняння кодеків за якістю і стисненням буде проведене пізніше.

На даному етапі доцільно скласти таблицю зі значеннями середнього навантаження на процесор у відсотках в процесі виконання задачі

Таблиця 3.2

Задача	H.264	H.265	MJPEG	MPEG 4 Part 2	ProRes	VP9
Транскодування	74	89	50	43	95	76
Зміна FPS	71	87	47	42	82	67
Зміна роздільної здатності	90	82	50	54	75	60
Додання водяного знаку	71	89	57	60	96	67

Найбільші значення навантаження можна помітити для кодеків H.264, H.265 і ProRes.

Поява H.265 серед кодеків, які проявили себе найскладнішими для обробки не дивує. По кількості проблем з відео, отриманими в результаті було помітно що процесор не встигає обробити всі кадри. H.264 в даній ситуації мав би аналогічні проблеми, якби навантаження було більшим хоча б на 6-8%, адже принцип роботи цих відеокодеків є досить схожим.

Кодеки що використовують inter-frame compression очікувано проявили себе краще в плані використання процесора, адже потребують меншої

кількості обчислень, окрім ProRes. Даний кодек продукує файли настільки значних розмірів, що саме розміри є причиною стількох проблем.

Найцікавішими є показники кодека VP9. При такій кількості артефактів не було б нічого дивного у навантаженні на процесор аналогічному до H.265, але його не спостерігається. Тобто кодек не зміг використати повний потенціал процесора, через що відео з його застосуванням виглядають найгірше.

На це є декілька причин. Перша з них це те що процесор AMD Ryzen 5 2500U, який використовується для тестування, не підтримує апаратне прискорення для даного кодеку. Таким чином, кодек може використовуватись, але не настільки ефективно як міг би, що й доводять показники використання процесора. Другою причиною є недостатня оптимізованість кодеку VP9, а саме те, що він схильний неефективно працювати з потоками, не використовуючи всі з них.

Тепер варто провести аналогічне порівняння для 2-го набору вхідних даних, тобто зі зниженою роздільною здатністю. Це дасть другий шанс кодекам, що не змогли проявити себе раніше

З транскодуванням, як і раніше, не впорався кодек VP9. Було отримано відео тривалістю 17 секунд з значною кількістю статичних кадрів і графічних артефактів.

Ситуація зі зміною FPS аналогічна

Після зміни роздільної здатності кардинальних змін також не було, але відеокодек VP9 показав трохи краще зображення. Артефакти і статичні кадри все ще присутні у значній кількості, однак можна однозначно сказати що стало краще

Накладання водяного знаку також не відрізнялось від попередніх тестів.

Порівняльна таблиця використання процесора виглядає наступним чином:

Таблиця 3.3

Задача	H.264	H.265	MJPEG	MPEG 4 Part 2	ProRes	VP9
Транскодування	48	61	35	30	54	48
Зміна FPS	36	53	29	30	45	47
Зміна роздільної здатності	24	55	29	31	43	42
Додання водяного знаку	45	70	39	34	64	50

Дана таблиця знову демонструє складність використання кодеку VP9 без апаратного прискорення, адже з такими значеннями навантаження на процесор все точно мало б бути гаразд, але ні. Інші значення не є настільки цікавими, адже кодек VP9 був єдиним, який мав проблеми при використанні даного набору вхідних даних.

Слід переходити до третього джерела відео, тобто MJPEG 1280x720 10 FPS з USB-камери

Єдина помітна проблема з даним джерелом під час транскодування проявилась з відеокодеком VP9. Було помічене зниження кількості кадрів на секунду, але не надто значне. Також були помітні легкі графічні артефакти, що не сильно шкодять відео У порівнянні з показниками цього кодека раніше це значне покращення результатів. Але й інші кодеки проявили себе добре і не викликали до себе питань.

Також було помічено артефакт запису відео, причиною якого однозначно є сама камераЦей артефакт вимагає більш подробного опису. Для тестування запису відео з USB-камери було створено статичний кадр зі змінним джерелом світла, яке вмикалось і вимикалось декілька разів протягом секунди. Увімкнене джерело світла в кадрі продемонстроване на рис.3.6., вимкнене на рис.3.7, а артефакт, який є комбінацією кадрів з увімкненим і вимкненим світлом на рис.3.8. На нього варто звернути увагу, щоб у майбутньому не прийняти його за артефакт якогось конкретного кодеку

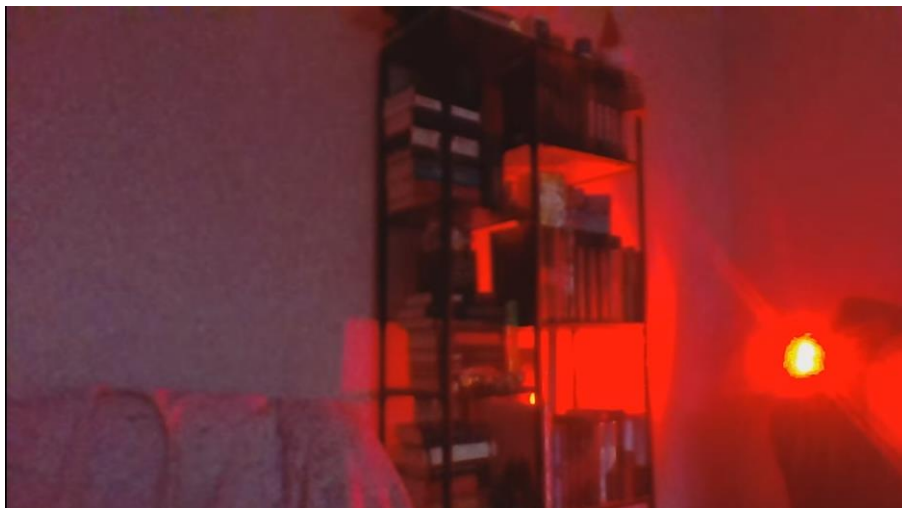


Рис. 3.4. Додаткове джерело світла увімкнене

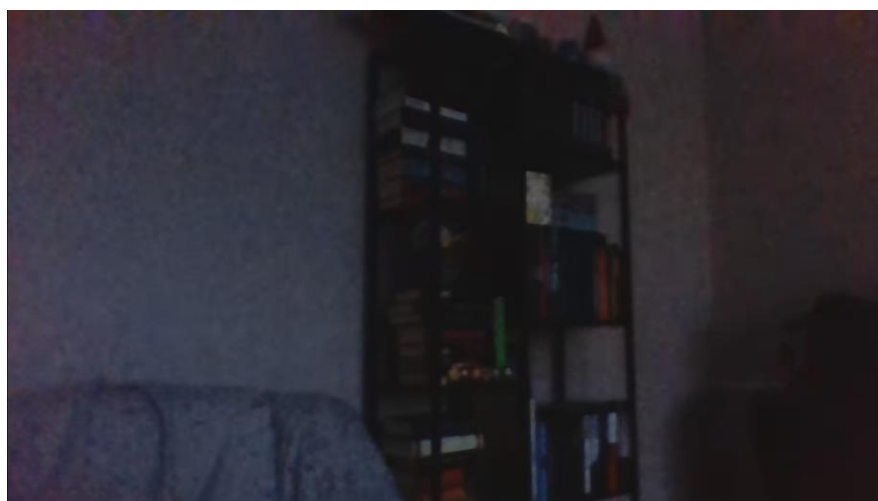


Рис. 3.5. Додаткове джерело світла вимкнене, об'єкти освітлені слабо

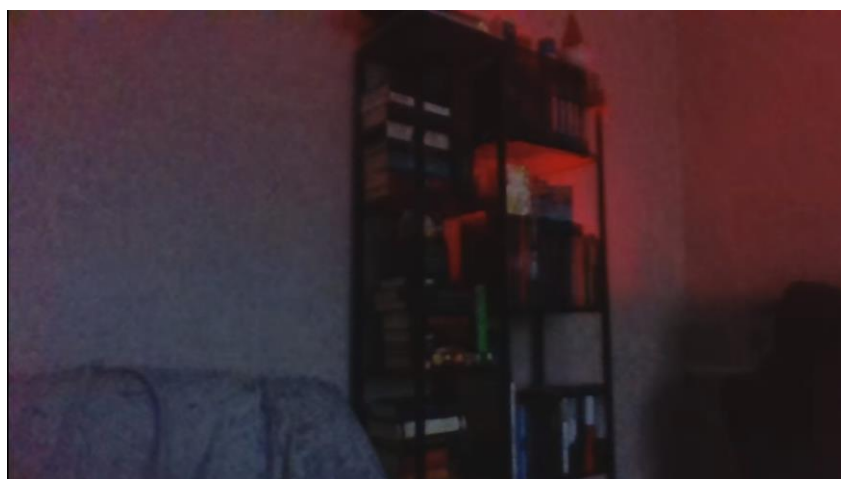


Рис. 3.6. Додаткове джерело світла вимкнене, а світло на стіні є.

Під час зміни кількості кадрів на секунду також були помітні легкі проблеми з відеокодеком VP9, аналогічні описаним вище

Те саме стосується зміни роздільної здатності

Додання водяного знаку не було проблемною задачею для жодного з кодеків, окрім VP9, який знову не зміг закодувати відео утримавши вхідну кількість кадрів на секунду.

Таблиця середніх значень завантаженості процесора під час виконання операції:

Таблиця 3.4

Задача	H.264	H.265	MJPEG	MPEG 4 Part 2	ProRes	VP9
Транскодування	10	53	7	8	27	39
Зміна FPS	10	36	7	6	19	25
Зміна роздільної здатності	8	10	14	10	16	23
Додання водяного знаку	24	48	12	13	31	34

Наступним інструментом, результати якого буде досліджено є FluentFfmpeg.

Для першого варіанту вхідного відео, тобто RTSP-потік, 1280x720, 25 FPS, і кодеку H.264 можна зробити такі висновки

При транскодуванні графічні артефакти і статичні кадри були наявні у результатах роботи кодеків H.265, ProRes і VP9. Найменше їх було відео з кодеком ProRes, найбільше у відео з кодеком VP9, яке, крім того, замість 20-24 секунд триває 13.

При зміні FPS не спостерігається проблем у відео з кодеком ProRes. Яскравий приклад того, що зміна частоти кадрів може служити дієвим інструментом оптимізації обробки відео. Інші кодеки не продемонстрували відчутних змін у своїй роботі

При зміні роздільної здатності єдиний кодек, відео якого містить проблеми це VP9. Артефакти дуже регулярні, і мають доволі значний характер (рис. 3.9)



Рис. 3.7. Графічні артефакти відеокодека VP9

Додавання анімованого водяного знаку є проблемним для кодеків H.264, H.265, ProRes і VP9, в цілому аналогічно до виконання цієї задачі за допомогою Ffmpeg

Складемо таблицю використання процесора:

Таблиця 3.5

Задача	H.264	H.265	MJPEG	MPEG 4 Part 2	ProRes	VP9
Транскодування	78	91	49	44	97	75
Зміна FPS	61	93	48	47	86	74
Зміна роздільної здатності	63	79	53	48	74	59
Додання водяного знаку	93	91	54	51	96	73

Отримана таблиця в основному збігається з аналогічно таблицею для звичайного Ffmpeg. В середньому відсоток використання процесора є трохи вищим ніж при використанні Ffmpeg у чистому вигляді, оскільки дана надбудова може забирати на себе частину продуктивності

Далі слід проаналізувати дані, отримані в результаті виконання операції з другим набором вхідних даних, тобто зі зниженою до 854x480 роздільною здатністю

При транскодуванні були присутні артефакти на відео з кодеком H.265 (рис. 3.8). Вони не мали настільки значний характер, але проявлялись неодноразово



Рис. 3.8. Графічні артефакти кодека H.265

Найбільшу кількість графічних артефактів очікувано мав кодек VP9. (рис.3.9)



Рис. 3.9. Графічні артефакти кодека VP9

Інші кодеки показали себе добре і не створили проблем в результаті своєї роботи

При зміні кадрової частоти єдиним кодеком, який не впорався з задачею на належному рівні був VP9.

Зміна роздільної здатності призвела до аналогічних результатів

Процес накладання водяного знаку також викликав появу артефактів на відео з кодеком VP9, але, більш цікавим був результат для кодека H.265. Графічні артефакти і статичні кадри були фактично відсутні, хоча для транскодування, яке є простішою задачею, подібні проблеми були характерні. З цього можна зробити висновок що кодек H.265 використовує наявні ресурси на максимум, а отже неможливо чітко передбачити чи буде операція виконана якісно, адже навіть невелике збільшення навантаження призведе до появи проблем з відео, яке буде отримане в результаті.

Для того щоб звірити даний висновок з реальними показниками, а також оцінити загальну картину використання ресурсів складемо порівняльну таблицю використання процесора:

Таблиця 3.6

Задача	H.264	H.265	MJPEG	MPEG 4 Part 2	ProRes	VP9
Транскодування	39	80	28	35	52	48
Зміна FPS	43	52	30	30	49	48
Зміна роздільної здатності	39	40	33	30	37	44
Додання водяного знаку	48	75	34	35	59	46

Отримана таблиця дає відповідь на питання про артефакти з кодеком H.265. Імовірно, сама система на якій відбувалось виконання операції, в цей момент навантажила чимось процесор, адже така метрика однозначно вибивається з вже показаної раніше закономірності. Процес транскодування є базовим і ніяк не може бути складнішим за додавання водяного знаку до відео. Імовірно, було наявне деяке додаткове навантаження, яке доволі швидко зникло

Було вирішено провести додаткове тестування транскодування з застосуванням кодеку H.265. Результатом стало чисте відео без графічних артефактів і середня використання процесору на рівні 57%, що добре вписується в загальну статистику. Отже результат, отриманий раніше є хибним. В усьому іншому таблиця знову ж дуже подібна до отриманої раніше за допомогою Ffmpeg.

Наступним і останнім для даного інструменту кроком буде аналіз даних для третього набору вхідних даних, тобто USB-камера, 1280x720, 10 FPS, MJPEG

Процес транскодування не викликав значних проблем, єдина проблема це періодичне зниження кадрової частоти для кодека VP9

Процеси зміни FPS а також роздільної здатності мали аналогічні результати

Процес накладання водяного знаку також призвів до зниження кадрової частоти відео з кодеком VP9.

Середні значення використання процесора вказано у таблиці:

Таблиця 3.7

Задача	H.264	H.265	MJPEG	MPEG 4 Part 2	ProRes	VP9
Транскодування	24	44	9	5	26	41
Зміна FPS	15	35	10	5	16	27
Зміна роздільної здатності	16	31	8	7	20	23
Додання водяного знаку	28	54	14	12	30	35

З таблиці очевидно що жодних проблем виникнути не мало б, але вони все ж виникли з кодеком VP9 з описаних раніше причин

Останнім інструментом, результати якого буде досліджено є Gstreamer.

Для першого варіанту вхідного відео, тобто RTSP-потік, 1280x720, 25 FPS, і кодеку H.264 можна зробити такі висновки

Перше на що слід звернути увагу це стабільна присутність графічних артефактів на початку відео. Відбувається це через різний принцип роботи Ffmpeg і Gstreamer з потоковим відео яке використовує кодеки з inter-framer compression. Ffmpeg чекає надсилання I-frame, тоді як Gstreamer може почати відео з P-frame, що викличе графічні артефакти аж до моменту появи першого I-frame. Це не є критичним, адже ffmpeg просто відкинув би цей шматок відео, або зробив би його статичним кадром на 1-2 секунди.



Рис. 3.10. Графічні артефакти при запуску відео обробленого за допомогою Gstreamer

При транскодуванні Gstreamer показав себе дуже ефективно. Графічних артефактів чи подібних проблем не було помічено на жодному з відео. Однак, якість відео з кодеками H.265 і VP9 була нижчою, а для кодеків MJPEG і MPEG 4 Part 2 – вищою. Це відбувається через те, що Gstreamer має інші стандартні налаштування якості зображення. Крім того, Gstreamer має функцію підлаштування якості під можливості системи. Саме цим можна пояснити відсутність артефактів при використанні кодеків, які викликали їх появу раніше і вищу якість при використанні кодеків, які раніше давали доволі неякісну картинку

Стосовно таких задач як зміна роздільної здатності і зміна FPS проблем також помічено не було

З додаванням водяного знаку виникли проблеми. Gstreamer в цілому погано підтримує подібні маніпуляції з відео, оскільки його функціонал є більш обмеженим ніж функціонал Ffmpeg. Виконати настільки ж ефективно додавання водяного знаку не вдалося, з отриманих відео жодне не відповідало всім заданим критеріям. Розберемо кожен з кодеків по порядку:

H.264. Відео без проблем але тривалість лише 8 секунд.

H.265. Відео без проблем, тривалість 4 секунди.

MJPEG. Водяний знак зник через 5 секунд, тривалість відео в нормі, інші проблеми відсутні.

Кодеки MPEG 4 Part 2 і ProRes мають проблеми оскільки плагіни Gstreamer не підтримують alpha-канал, і спроба накладання водяного знаку створює пошкоджені відео файли.

VP9. Також пошкоджений файл, оскільки плагін для VP9 не підтримує дану операцію.

Середні значення навантаження процесора у відсотках вказані в таблиці:

Таблиця 3.8

Задача	H.264	H.265	MJPEG	MPEG 4 Part 2	ProRes	VP9
Транскодування	92	99	99	99	99	99
Зміна FPS	91	99	98	46	61	72
Зміна роздільної здатності	93	85	53	43	53	63
Додання водяного знаку	92	94	50	-	-	-

З таблиці очевидно що Gstreamer намагався максимально використовувати наявні ресурси системи.

Для другого набору вхідних даних не було отримано значних проблем з якістю відео, оскільки другий набір простіший в обробці ніж перший, який теж не викликав жодних проблем з транскодуванням, зміною FPS і зміною

роздільної здатності. Однак, накладання водяного знаку дало дещо інші результати.

H.264. Тривалість відео була в нормі, однак водяний знак зник через 6 секунд, а також саме відео містить значну кількість статичних кадрів. Графічних артефактів не було.

H.265. Проблеми аналогічні до попереднього кодексу, але тривалість 14 секунд

MJPEG має аналогічні проблеми що і для попереднього набору даних

Кодеки MPEG 4 Part 2, ProRes, VP9 створили пошкоджені файли, що не можуть бути відтворені.

Середні значення навантаження процесора у відсотках вказані в таблиці:

Таблиця 3.9

Задача	H.264	H.265	MJPEG	MPEG 4 Part 2	ProRes	VP9
Транскодування	91	72	45	28	34	50
Зміна FPS	81	65	32	35	29	49
Зміна роздільної здатності	23	54	33	29	27	41
Додання водяного знаку	88	63	32	-	-	-

Помітно що бібліотека Gstreamer ставить собі за мету використати ресурси системи на максимум. 23% при зміні роздільної здатності на кодеку H.264 має свою причину. Дуже складно навантажити систему, оброблюючи відео шириною 427 і висотою 240 пікселів за допомогою такого відносно легкого кодеку як H.264

При аналізі даних, що були отримані в результаті роботи з третім набором вхідних даних, було помічено проблеми з кількістю кадрів на секунду. Ця проблема була наявна для кожної з операцій і таких кодеків як H.264, H.265, VP9.

Що стосується інших проблем, то процес додавання водяного знаку викликав вже описані вище проблеми з тривалістю, причому ще більш значні

ніж раніше, крім того, кодек MJPEG цього разу теж мав зниження FPS, що раніше не зустрічалося

Таблиця 3.10

Задача	H.264	H.265	MJPEG	MPEG 4 Part 2	ProRes	VP9
Транскодування	71	18	17	6	11	36
Зміна FPS	50	19	3	3	11	37
Зміна роздільної здатності	49	24	7	8	7	19
Додання водяного знаку	75	15	10	-	-	-

Можна зробити висновок, що Gstreamer має проблеми з вхідним кодеком MJPEG. Ці проблеми дуже серйозні

Слід також відповісти на питання якості зображення, а саме як оцінити якість зображення для кожного з кодеків. Почати варто з Ffmpeg та FluentFfmpeg, оскільки вони мають однакові налаштування якості за замовчуванням.

Кодек з найкращою якістю це однозначно ProRes.



Рис. 3. 11. Демонстрація якості зображення для кодека ProRes

Найгіршу якість серед представлених кодеків надає MPEG 4 Part 2



Рис. 3.12. Демонстрація якості зображення для кодека MPEG 4 Part 2

Доцільно вважати якість, яку надає ProRes дуже високою, а якість яку надає MPEG 4 Part 2 дуже низькою. Таким чином можна буде описати якість відео з іншими відеокодеками.

Таблиця 3.11

Кодек	Якість
H.264	Висока
H.265	Висока
MJPEG	Низька
MPEG 4 Part 2	Дуже низька
ProRes	Дуже висока
VP9	Висока

Також варто зауважити що Ffmpeg, FluentFfmpeg, і Gstreamer дозволяють керувати якістю, що вже було відмічено при порівнянні зображення з кодеком MPEG-4 Part 2 в результаті транскодування за допомогою Ffmpeg/FluentFfmpeg і Gstreamer. В той самий час, Gstreamer

знизив якість для VP9, щоб результат не мав графічних артефактів або пропусків кадрів.

Також слід порівняти якість відео при транскодуванні з застосуванням Gstreamer. В цілому кодеки ProRes і H.264, показали себе приблизно однаково в плані якості.



Рис. 3.13. Якість зображення при використанні кодека ProRes
Найгіршу якість зображення мало з кодеком VP9



Рис. 3.14. Якість зображення для кодека VP9

Таблиця 3.12

Кодек	Якість
H.264	Висока
H.265	Висока
MJPEG	Середня
MPEG 4 Part 2	Висока
ProRes	Висока
VP9	Низька

На даному етапі можна скласти порівняльні таблиці розмірів і тривалості відео різних кодеків залежно від вхідних даних. Порівнювати кожен з процесів немає сенсу, адже загальна картина виглядатиме так аналогічно, тому буде виконане порівняння на прикладі базового процесу-транскодування

Дана таблиця ілюструє ефективність стиснення за допомогою різних відеокодеків. Чим менший файл тим краще стиснення [36], але зачасту це супроводжується або складнішим процесом обробки або втратою якості зображення.

Таблиця для Ffmpeg/FluentFfmpeg

Таблиця 3.13

Кодек	RTSP, 1280x720		RTSP, 854x480		USB, 1280x720	
	Розмір (КБ)	Тривалість (секунди)	Розмір (МБ)	Тривалість (секунди)	Розмір (МБ)	Тривалість (секунди)
H.264	3166	22	2316	20	6499	24
H.265	1024	19	747	24	1706	24
MJPEG	10038	24	5128	24	5163	24
MPEG 4 Part 2	2486	24	1240	24	2298	24
ProRes	192160	23	120384	24	113657	24
VP9	914	12	743	17	1746	19

Кодек	RTSP, 1280x720		RTSP, 854x480		USB, 1280x720	
	Розмір (КБ)	Тривалість (секунди)	Розмір (МБ)	Тривалість (секунди)	Розмір (МБ)	Тривалість (секунди)
H.264	5335	23	5071	23	4049	22
H.265	6066	23	5618	23	6108	24
MJPEG	29346	23	19768	23	17927	24
MPEG 4 Part 2	13833	23	7530	23	11796	24
ProRes	95246	23	55392	23	53128	23
VP9	668	23	673	23	637	22

З цих таблиць можна зробити наступні висновки:

Gstreamer знизив не тільки якість, а й ступінь стиснення для кодеку H.265. Алгоритм стиснення цього кодека є досконалішим ніж у H.264 [37], але розміри файлів демонструють зворотнє.

Також обидві таблиці демонструють очевидну перевагу кодеків з inter-frame compression. Вони здатні забезпечити гарну якість і сильне стиснення водночас. Вони більше навантажують процесор, але іноді ця різниця не настільки критична. Наприклад, при використанні Ffmpeg або FluentFfmpeg, для RTSP H.264 відео з роздільною здатністю 854x480, навантаження на процесор при транскодуванні назад в H.264 є лише на 5-10% більшим ніж при використанні MJPEG і MPEG-4 Part 2, забезпечуючи при цьому значно кращу якість і розмір файлу значно менший за MJPEG і трохи більший за MPEG-4 Part 2.

Але кодеки з intra-frame compression також мають перевагу, що може стати критичною в певних ситуаціях. Вони не страждають від втрати кадрів. Фактично, єдине що зміниться якщо один з кадрів буде відсутнім це лише зниження кадрової частоти. В свою чергу, кодеки які використовують inter-frame compression одразу починають продукувати графічні артефакти у разі втрати кадру.

Більшість кодеків є досить гнучкими в налаштуванні і за потреби можуть бути підігнані під наявні технічні можливості. Єдиний кодек який погано себе проявив в додаткових налаштуваннях це ProRes. Він добре справляється зі зберіганням файлів без втрат або з мінімальними втратами, але у разі більшого зниження якості вона втрачається досить швидко, а розмір лишається все ще значним.

Так, в контексті подальшої передачі даних у вигляді потоку відео кінцевий розмір файлу не відіграє ролі. Однак, чим більшим є розмір файлу, тим більшим є розмір кадру, а значить його передача може бути важчою. В основному, це стосується кодеку ProRes, адже розміри файлів при його використанні значно більші ніж у інших кодеків, що може викликати серйозні проблеми при подальшій обробці.

Відеокодек VP9 також можна відкинути, адже його неможливо використовувати з тою ж ефективністю що й його аналоги якщо процесор його не підтримує [38]. Апаратне прискорення для даного кодеку достатньо широко підтримується сучасними десктопними процесорами, однак цього не можна сказати про слабші процесори, наприклад процесори, що використовуються для створення мініатюрних або взагалі одноплатних [39] комп'ютерів, а вони доволі широко застосовуються для роботи з потоком відео

Якщо є потреба передавати комунікаційним каналом якомога меншу кількість даних, то найбільш доцільно використовувати H.265, що, на жаль, сильніше навантажуватиме систему

Якщо хоча б один з комунікаційних каналів є нестабільним, або якщо робота навантажує процесор на близькі до максимальних значення, доцільніше буде використовувати MJPEG або MPEG 4 Part 2. Вони гарно себе проявлять навіть при відсутності декількох кадрів, і не є складними в обробці [40].

Якщо йдеться про стабільний і надійний комунікаційний канал то найкращим вибором буде кодек H.264. Він забезпечує найкраще співвідношення складності в обробці до якості зображення, крім того, має

гарне стиснення і, в силу свого віку, підтримується всіма сучасними системами

Ffmpeg проявив себе як більш надійний і простий в налаштуванні інструмент. Він не вимагає великої кількості параметрів для роботи, є дуже гнучким, а також забезпечує простіший пошук несправностей у разі їх наявності. Також, він є простішим у встановленні і налаштуванні і працює стабільно незалежно від джерела відео.

Fluent-ffmpeg має усі чесноти стандартного Ffmpeg, але спрощує роботу з ним на платформі Node.js. В середньому має на 3-5% більше використання процесора у порівнянні зі звичайним Ffmpeg, що в контексті виконаного дослідження не було помітною різницею, але таку різницю варто брати до уваги. Інших переваг, окрім спрощення роботи, не надає.

Gstreamer є складнішим в освоєнні, незважаючи на свою властивість адаптуватись до можливостей системи. Крім того, він має досить обмежений функціонал, не надає можливості повноцінно працювати з відео що надходить з USB-камери з кодеком MJPEG, і не здатний накласти водяний знак, витрачаючи на це більше ресурсів ніж Ffmpeg і FluentFfmpeg. Водночас, він добре проявив себе при створенні відео з H.264 RTSP-потоків, витримавши баланс між можливостями системи, якістю і стисненням.

Висновки до третього розділу

Були виконані всі заплановані операції. Після цього був проведений комплексний аналіз отриманих результатів. Для початку, результати були перевірені на предмет графічних артефактів візуально

Потім було складено таблиці з середніми значенням навантаження на процесор під час виконання операцій. Було також складено таблиці з якістю отриманих відео залежно від кодеку та інструменту

Було проаналізовано ступінь стиснення і втрати якості при використанні різних кодеків та інструментів, і сформульовано рекомендації щодо умов, в яких досліджені кодеки зможуть проявити себе найкраще, а також щодо правильного використання досліджених інструментів.

ВИСНОВКИ

Під час виконання кваліфікаційної роботи магістра було досліджено предметну область, спроектовано та створено програмне забезпечення, що являє собою середовище для автоматичного виконання задач і збору інформації щодо обробки потокового відео, і проведено аналіз отриманих даних, для чого було вирішено такі завдання.

Виконано дослідження і аналіз предметної області, а саме проаналізовано процес транскодування відео, описано різницю між підходами до кодування відео і основними кодеками. Також було проаналізовано існуючі інструменти для обробки потокового відео для подальшого дослідження і обґрунтовано вибір мови програмування.

Було виконане дослідження реалізації базових процесів обробки потокового відео з застосуванням існуючих інструментів, реалізовано транскодування для обраної мови програмування з застосуванням всіх обраних інструментів, з подальшим розширенням і доданням таких операцій як зміна кількості кадрів на секунду, зміна роздільної здатності і додання водяного знаку.

Був виконаний детальний візуальний аналіз виконання задач кожним з інструментів для кожного з цільових кодеків, маючи три різних варіанти вхідних даних. Також був проведений аналіз метрик використання процесора що збирались для кожної з операцій індивідуально. Були також порівняні такі параметри як фінальний розмір файлів для розуміння ефективності стиснення кожного з відеокодеків і сформовано висновки щодо ефективності і доцільності використання обраних інструментів та кодеків

Проведене дослідження надає значну кількість точної інформації, що буде корисною при розробці проєктів, що включають в себе обробку потокового відео, і стане особливо важливим при проєктуванні системи яка включає обробку відео використовуючи мініатюрні або одноплатні комп'ютери.

Протягом дослідження джерел інформації, аналізу технологій, розробки програмного забезпечення було відточено навички роботи з досліджуваними інструментами, отримано нові знання, навички та вміння з пошуку інформації, відбору достовірних джерел, розробки автоматизованої системи для спрощення процесу збору даних, а також зі сфери обробки відео і його основних принципів в цілому. Отримано розуміння переваг та недоліків кожного з досліджених відеокодеків і кожного з досліджених інструментів для роботи з відео, визначено найбільш ефективні кодеки для конкретних задач і дано чітку характеристику інструментам щодо їх ефективності, зручності і функціоналу.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Definition. Video streaming [Електронний ресурс] / Alexander S. Gillis //TechTarget – 2021р. – Режим доступу до ресурсу: <https://www.techtarget.com/searchunifiedcommunications/definition/streaming-video>
2. 5 Node.js Advantages and Disadvantages and What They Mean for Your Project [Електронний ресурс] / EPAM Editorial Team // EPAM – 2024 – Режим доступу до ресурсу <https://www.epam.com/careers/blog/5-node-js-advantages-and-disadvantages-and-what-they-mean-for-your-project>
3. Mastering Microservices Architecture with Node.js [Електронний ресурс] / Ayush Nandanwar // Medium – 2024 – Режим доступу до ресурсу: <https://medium.com/@ayushnandanwar003/mastering-microservices-architecture-with-node-js-db82cdb07b17>
4. Harnessing the Power of Node.js for Image and Video Processing [Електронний ресурс] / Folasayo Samuel Olayemi // DEV – 2023 – Режим доступу до ресурсу: https://dev.to/saint_vandora/harnessing-the-power-of-nodejs-for-image-and-video-processing-1ked
5. How and Why You Should Build Internet of Things Devices with Node.js [Електронний ресурс] / Oleh Romanyuk / / FreeCodeCamp – 2020 – Режим доступу до ресурсу: <https://www.freecodecamp.org/news/iot-and-node-js-how-to-catch-the-opportunity/>
6. Video codec: here's what you need to know [Електронний ресурс] / Phil Henken // Kaltura blog – 2024 – Режим доступу до ресурсу: <https://corp.kaltura.com/blog/video-codec/>
7. Compression artifacts [Електронний ресурс] / Cloudinary – 2024 – Режим доступу до ресурсу: <https://cloudinary.com/glossary/compression-artifacts>
8. What is image processing [Електронний ресурс] / GeeksForGeeks – 2024 – Режим доступу до ресурсу <https://www.geeksforgeeks.org/what-is-image-processing/>

9. What is transcoding? and why is it important for video streaming?
[Электронный ресурс] / Phill Henken // Kaltura blog – 2024 – Режим доступа до ресурсу: <https://corp.kaltura.com/blog/what-is-transcoding/>
10. Demystifying video transcoding: Importance, tools, types & best practices
[Электронный ресурс] / Rahul Nanwani // Imagekit.io – 2023 – Режим доступа до ресурсу: <https://imagekit.io/blog/video-transcoding/>
11. A Detailed Overview Of Popular Video Compression Techniques
[Электронный ресурс] / Vignesh.S.Krishnan // Imagekit.io – 2024 – Режим доступа до ресурсу: <https://imagekit.io/blog/video-compression-techniques/>
12. Difference between Inter and Intra Frame Compression [Электронный ресурс] / GeeksForGeeks – 2021 – Режим доступа до ресурсу: <https://www.geeksforgeeks.org/difference-between-inter-and-intra-frame-compression/>
13. JPEG files. [Электронный ресурс] / Adobe – 2024 – Режим доступа до ресурсу: <https://www.adobe.com/creativecloud/file-types/image/raster/jpeg-file.html>
14. Difference between Lossy Compression and Lossless Compression
[Электронный ресурс] / GeeksForGeeks – 2024 – Режим доступа до ресурсу: <https://www.geeksforgeeks.org/difference-between-lossy-compression-and-lossless-compression/>
15. Understanding video inter-frame compression [Электронный ресурс] / Vijaysing Prakash Patil // FastPix – 2024 – Режим доступа до ресурсу: <https://www.fastpix.io/blog/understanding-video-inter-frame-compression>
16. Realtime Video Stream Analysis with Computer Vision [Электронный ресурс] / Leo Ueno // Roboflow – 2024 – Режим доступа до ресурсу: <https://blog.roboflow.com/video-stream-analysis/>
17. How Does Amazon Go Work? Exploring Self-Serve Stores and the Technology Behind Them [Электронный ресурс] / Woz U // Woz U – 2022 – Режим доступа до ресурсу: <https://woz-u.com/blog/amazon-go-work-exploring-self-serve-stores-technology/>

18. What is Video Resolution, Types, Advantages & Disadvantages [Электронный ресурс] / VdoCipher – 2023 Режим доступа до ресурсу: <https://www.vdocipher.com/blog/video-resolution/>

19. Frame Rate: A Beginner's Guide [Электронный ресурс] / Doug Brunner // TechSmith – 2024 – Режим доступа до ресурсу: <https://www.techsmith.com/blog/frame-rate-beginners-guide/?srsltid=AfmBOorsCcmv1JpElPh76KYF7E87IqBOI6-ovNduiqkMKOIOmgfPaa>

20. H.265 Video Codec For Surveillance Systems: Why Use It? [Электронный ресурс] / Anish Devasia // Safe and Sound Security – 2024 – Режим доступа до ресурсу: <https://getsafeandsound.com/blog/h-265-video-codec/>

21. What is H.264 video encoding? How does H.264 codec work? [Электронный ресурс] / Prabu Kumar // e-con Systems – 2023 – Режим доступа до ресурсу <https://www.e-consystems.com/blog/camera/technology/what-is-h-264-video-encoding-how-does-h-264-codec-work/>

22. A Comprehensive Guide on H.265 Codec [Электронный ресурс] / Shashikanth Manjunatha // Gumlet – 2024 – Режим доступа до ресурсу: <https://www.gumlet.com/learn/what-is-h265/>

23. Everything you need to know about VP9 codec [Электронный ресурс] / Many Chaudhary // Imagekit.io – 2024 – Режим доступа до ресурсу: <https://imagekit.io/blog/vp9-codec/>

24. AV1 Codec - Complete guide for video application devs [Электронный ресурс] / Abhinav Rituraj // Imagekit.io – 2024 – Режим доступа до ресурсу: <https://imagekit.io/blog/av1-codec/>

25. What is ProRes and why does Apple keep mentioning it? [Электронный ресурс] / Adrian Kingsley-Hughes // zdnet – 2023 – Режим доступа до ресурсу: <https://www.zdnet.com/article/what-is-prores-and-why-does-apple-keep-mentioning-it/>

26. MPEG-4 Part 2 [Электронный ресурс] / Margaret Rouse // Technopedia – 2020 – Режим доступа до ресурсу: <https://www.techopedia.com/definition/10756/mpeg-4-part-2>
27. What is Motion JPEG (MJPEG) Video Codec [Электронный ресурс] / Abhishek Ghosh // The customize windows – 2024 – Режим доступа до ресурсу: <https://thecustomizewindows.com/2024/05/what-is-motion-jpeg-mjpeg-video-codec/>
28. Mastering Asynchronous Programming in Node.js [Электронный ресурс] / Vishwas Acharya // Medium – 2023 – Режим доступа до ресурсу: <https://medium.com/@vishwasacharya/mastering-asynchronous-programming-in-node-js-174cb75272a0>
29. NodeJS vs Python: Which one to choose for 2024 [Электронный ресурс] / A Smith // Medium – 2024 – Режим доступа до ресурсу: <https://medium.com/front-end-weekly/nodejs-vs-python-which-one-to-choose-for-2024-0477d3ab7d5a>
30. JavaScript vs. TypeScript: What's the difference? [Электронный ресурс] / Cameron McKenzie // TheServerSide – 2023 – Режим доступа до ресурсу: <https://www.theserverside.com/tip/JavaScript-vs-TypeScript-Whats-the-difference>
31. What is FFmpeg? [Электронный ресурс] / Sristi Chowdhury // Medium – 2022 – Режим доступа до ресурсу: <https://medium.com/weekly-webtips/what-is-ffmpeg-a2ba2afbd254>
32. The Unbelievable Capabilities of FFmpeg — See What You’ve Been Missing! [Электронный ресурс] / Jainath Ponnala // Medium – 2023 – Режим доступа до ресурсу: <https://ponnala.medium.com/the-unbelievable-capabilities-of-ffmpeg-see-what-youve-been-missing-59a10f18ec1f>
33. Introduction to GStreamer for beginners — part 1 [Электронный ресурс] / Raktim Bora // Medium – 2023 – Режим доступа до ресурсу: <https://medium.com/@reachraktim/introduction-to-gstreamer-for-beginners-part-1-fe85342cacb8>

34. Introduction of GStreamer [Электронный ресурс] / Ido Magor // Medium – 2020 – Режим доступа до ресурсу: <https://idomagor.medium.com/introduction-of-gstreamer-8a513e5478d0>

35. FFmpeg Codecs Documentation [Электронный ресурс] FFmpeg Developers // FFmpeg – 2024 – Режим доступа до ресурсу: <https://www.ffmpeg.org/ffmpeg-codecs.html>

36. Developer Network Series: Everything you need to know about Video Compression [Электронный ресурс] / Andrea Fassinà // Bitmovin – 2020 – Режим доступа до ресурсу: <https://bitmovin.com/blog/video-compression-fundamentals/>

37. What is the difference between H.264 vs. H.265? [Электронный ресурс] / Arushi Gupta // api.video – 2023 – Режим доступа до ресурсу: <https://api.video/blog/video-trends/H264-vs-H265/>

38. VP9 Codec: The Complete Guide to Google's Open Source Video Codec [Электронный ресурс] / Andy Francis // Bitmovin – 2023 – Режим доступа до ресурсу: <https://bitmovin.com/blog/vp9-codec-status-quo/>

39. What is a Single-Board Computer (SBC)? [Электронный ресурс] / David Ruddock // Esper – 2024 – Режим доступа до ресурсу: <https://www.esper.io/blog/what-is-a-single-board-computer-sbc>

40. Choosing the Right Video Compression Format: A Comparison of H.264, H.265, and MJPEG [Электронный ресурс] / Christy Wong // Supertek – 2023 – Режим доступа до ресурсу: <https://www.supertekmodule.com/choosing-the-right-video-compression-format-a-comparison-of-h-264-h-265-and-mjpeg/>

ДОДАТКИ

Повний код програмного забезпечення :

videoModuleTester.ts

```

export enum VideoCodec {
    H264 = 'H264',
    H265 = 'H265',
    MJPEG = 'MJPEG',
    ProRes = 'ProRes',
    MPEG4_PART_2 = 'MPEG4_PART_2',
    VP9 = 'VP9',
}

export enum Resolution {
    Low240p = '427x240',
    LowForH265 = '428x240',
    HalfHD = '640x360',
    SD = '854x480',
    HD = '1280x720',
    FullHD = '1920x1080',
    QHD = '2560x1440',
}

export enum Task {
    TRANSCODE = 'TRANSCODE',
    RESIZE = 'RESIZE',
    SET_FPS = 'SET_FPS',
    WATERMARK = 'WATERMARK'
}

export abstract class VideoModuleTester {
    abstract getName(): string;
    abstract transcode(outputCodec: VideoCodec, resolution: Resolution): void;
    abstract resize(videoCodec: VideoCodec, outputResolution: Resolution): void;
    abstract setFps(outputCodec: VideoCodec, fps: number): void;
    abstract addWatermark(videoCodec: VideoCodec): void;
    abstract killProcess(): void;
}

```

simpleFfmpeg/index.ts

```

import {ChildProcessWithoutNullStreams, spawn} from 'child_process';
import path from 'path';

import {constants} from '../constants';

import {Resolution, Task, VideoCodec, VideoModuleTester} from '../videoModuleTester';
import {getFileName} from '../helpers';

export default class SimpleFfmpeg extends VideoModuleTester {
    private ffmpegProcess: ChildProcessWithoutNullStreams | undefined;

    getName(): string {
        return 'SimpleFfmpeg';
    }

    transcode(outputCodec: VideoCodec, resolution: Resolution): void {
        const { codec: inputCodec } =
            SimpleFfmpeg.getInitialVideoConfigForFfmpeg(constants.INPUT_CODEC.toString());
        const { codec, format } =
            SimpleFfmpeg.getInitialVideoConfigForFfmpeg(outputCodec);
        const filename = getFileName(Task.TRANSCODE, this.getName(), outputCodec,
            resolution);
        this.ffmpegProcess = spawn('ffmpeg', [
            '-i', constants.INPUT_STREAM,
            '-input_format', inputCodec,
            '-c:v', codec,
            '-f', format,

```

```

        `${constants.OUTPUT_FOLDER}/${filename}.${format}`,
    });
    this.addListeners();
}

resize(outputCodec: VideoCodec, outputResolution: Resolution): void {
    const { codec: inputCodec } =
SimpleFfmpeg.getInitialVideoConfigForFfmpeg(constants.INPUT_CODEC.toString());
    const { codec, format } =
SimpleFfmpeg.getInitialVideoConfigForFfmpeg(outputCodec);
    const filename = getFileName(Task.RESIZE, this.getName(), outputCodec,
outputResolution);
    let size = outputResolution;
    if (outputCodec === VideoCodec.H265) {
        size = Resolution.LowForH265;
    }
    this.ffmpegProcess = spawn('ffmpeg', [
        '-i', constants.INPUT_STREAM,
        '-input_format', inputCodec,
        '-vf', `scale=${size}`,
        '-c:v', codec,
        '-f', format,
        `${constants.OUTPUT_FOLDER}/${filename}.${format}`,
    ]);
    this.addListeners();
}

setFps(outputCodec: VideoCodec, fps: number): void {
    const { codec: inputCodec } =
SimpleFfmpeg.getInitialVideoConfigForFfmpeg(constants.INPUT_CODEC.toString());
    const { codec, format } =
SimpleFfmpeg.getInitialVideoConfigForFfmpeg(outputCodec);
    const filename = getFileName(Task.SET_FPS, this.getName(), outputCodec,
constants.INPUT_RESOLUTION.toString());
    this.ffmpegProcess = spawn('ffmpeg', [
        '-i', constants.INPUT_STREAM,
        '-input_format', inputCodec,
        '-r', fps.toString(),
        '-c:v', codec,
        '-f', format,
        `${constants.OUTPUT_FOLDER}/${filename}.${format}`,
    ]);
    this.addListeners();
}

addWatermark(videoCodec: VideoCodec): void {
    const { codec, format } =
SimpleFfmpeg.getInitialVideoConfigForFfmpeg(videoCodec);
    const filename = getFileName(Task.WATERMARK, this.getName(), videoCodec,
constants.INPUT_RESOLUTION.toString());
    const watermarkPath = path.join(process.cwd(), constants.WATERMARK_FILE);
    const ffmpegArgs = [
        '-i', constants.INPUT_STREAM,
        '-ignore_loop', '0',
        '-i', watermarkPath,
        '-filter_complex',
        '[1]format=rgba,colorchannelmixer=aa=0.8[logo];[0][logo]overlay=(W-
w)/2:(H-h)/2:format=auto,format=yuv420p',
        '-c:v', codec,
        '-f', format,
        `${constants.OUTPUT_FOLDER}/${filename}.mp4`,
    ];
    this.ffmpegProcess = spawn('ffmpeg', ffmpegArgs);
    this.addListeners();
}

private addListeners(): void {
    if (this.ffmpegProcess) {

```

```

        this.ffmpegProcess.stdout.on('data', (data) => {
            console.log(`FFmpeg output: ${data}`);
        });

        this.ffmpegProcess.stderr.on('data', (data) => {
            console.error(`FFmpeg error: ${data}`);
        });

        this.ffmpegProcess.on('close', (code) => {
            console.log(`FFmpeg process closed with code: ${code}`);
        });
    }

    static getInitialVideoConfigForFfmpeg(codec: string): { codec: string, format:
string } {
        switch (codec) {
            case VideoCodec.H264:
                return { codec: 'libx264', format: 'flv' };
            case VideoCodec.H265:
                return { codec: 'libx265', format: 'mp4' };
            case VideoCodec.MJPEG:
                return { codec: 'mjpeg', format: 'avi' };
            case VideoCodec.MPEG4_PART_2:
                return { codec: 'mpeg4', format: 'mp4' };
            case VideoCodec.ProRes:
                return { codec: 'prores_ks', format: 'mov' };
            case VideoCodec.VP9:
                return { codec: 'libvpx-vp9', format: 'webm' };
            default:
                return { codec: 'libx264', format: 'flv' };
        }
    }

    killProcess(): void {
        this.ffmpegProcess?.kill('SIGINT');
    }
}

```

fluentFfmpeg/index.ts

```

import ffmpeg from 'fluent-ffmpeg';
import path from 'path';

import { constants } from '../constants';

import { Resolution, Task, VideoCodec, VideoModuleTester } from
'../videoModuleTester';
import SimpleFfmpeg from '../simpleFfmpeg';
import { getFileName } from '../helpers';

export default class FluentFfmpeg extends VideoModuleTester {
    private ffmpeg: ffmpeg.FfmpegCommand | undefined;

    getName(): string {
        return 'FluentFfmpeg';
    }

    transcode(outputCodec: VideoCodec, resolution: Resolution): void {
        const { codec, format } =
SimpleFfmpeg.getInitialVideoConfigForFfmpeg(outputCodec);
        const filename = getFileName(Task.TRANSCODE, this.getName(), outputCodec,
resolution);
        this.ffmpeg = ffmpeg(constants.INPUT_STREAM)
            .output(`${constants.OUTPUT_FOLDER}/${filename}.${format}`)
            .videoCodec(codec)
            .format(format)
            .on('start', (commandLine) => {
                console.log('FFmpeg process started:', commandLine);
            })
            .on('end', () => {
                console.log('FFmpeg process finished.');
```

```

    })
    .on('error', (err, stdout, stderr) => {
        console.error('Error:', err);
        console.error('FFmpeg stderr:', stderr);
    });
    this.ffmpeg.run();
}

resize(outputCodec: VideoCodec, outputResolution: Resolution): void {
    const { codec, format } =
SimpleFfmpeg.getInitialVideoConfigForFfmpeg(outputCodec);
    const filename = getFileName(Task.RESIZE, this.getName(), outputCodec,
outputResolution);
    this.ffmpeg = ffmpeg(constants.INPUT_STREAM)
        .output(`${constants.OUTPUT_FOLDER}/${filename}.${format}`)
        .videoCodec(codec)
        .size(outputResolution)
        .format(format)
        .on('start', (commandLine) => {
            console.log('FFmpeg process started:', commandLine);
        })
        .on('end', () => {
            console.log('FFmpeg process finished.');
```

```

        .videoCodec(codec)
        .on('start', (commandLine) => {
            console.log('FFmpeg process started:', commandLine);
        })
        .on('end', () => {
            console.log('FFmpeg process finished.');
```

gstreamer/index.ts

```

import ffmpeg from 'fluent-ffmpeg';
import path from 'path';

import { constants } from '../constants';

import { Resolution, Task, VideoCodec, VideoModuleTester } from
'../videoModuleTester';
import SimpleFfmpeg from '../simpleFfmpeg';
import { getFileName } from '../helpers';

export default class FluentFfmpeg extends VideoModuleTester {
    private ffmpeg: ffmpeg.FfmpegCommand | undefined;

    getName(): string {
        return 'FluentFfmpeg';
    }

    transcode(outputCodec: VideoCodec, resolution: Resolution): void {
        const { codec, format } =
SimpleFfmpeg.getInitialVideoConfigForFfmpeg(outputCodec);
        const filename = getFileName(Task.TRANSCODE, this.getName(), outputCodec,
resolution);
        this.ffmpeg = ffmpeg(constants.INPUT_STREAM)
            .output(`${constants.OUTPUT_FOLDER}/${filename}.${format}`)
            .videoCodec(codec)
            .format(format)
            .on('start', (commandLine) => {
                console.log('FFmpeg process started:', commandLine);
            })
            .on('end', () => {
                console.log('FFmpeg process finished.');
```

```

    })
    .on('end', () => {
        console.log('FFmpeg process finished.');
```

```

    })
    .on('error', (err, stdout, stderr) => {
        console.error('Error:', err);
        console.error('FFmpeg stderr:', stderr);
    });
    this.ffmpeg.run();
}

setFps(outputCodec: VideoCodec, fps: number): void {
    const { codec, format } =
SimpleFfmpeg.getInitialVideoConfigForFfmpeg(outputCodec);
    const filename = getFileName(Task.SET_FPS, this.getName(), outputCodec,
constants.INPUT_RESOLUTION.toString());
    this.ffmpeg = ffmpeg(constants.INPUT_STREAM)
        .output(`${constants.OUTPUT_FOLDER}/${filename}.${format}`)
        .videoCodec(codec)
        .format(format)
        .outputOptions([
            '-r ${fps}'
        ])
    .on('start', (commandLine) => {
        console.log('FFmpeg process started:', commandLine);
    })
    .on('end', () => {
        console.log('FFmpeg process finished.');
```

```

    })
    .on('error', (err, stdout, stderr) => {
        console.error('Error:', err);
        console.error('FFmpeg stderr:', stderr);
    });
    this.ffmpeg.run();
}

addWatermark(videoCodec: VideoCodec): void {
    const { codec, format } =
SimpleFfmpeg.getInitialVideoConfigForFfmpeg(videoCodec);
    const filename = getFileName(Task.WATERMARK, this.getName(), videoCodec,
constants.INPUT_RESOLUTION.toString());
    const watermarkPath = path.join(process.cwd(), constants.WATERMARK_FILE);
    this.ffmpeg = ffmpeg(constants.INPUT_STREAM)
        .input(watermarkPath)
        .inputOptions('-ignore_loop 0')
        .output(`${constants.OUTPUT_FOLDER}/${filename}.${format}`)
        .complexFilter([
            '[1]format=rgba,colorchannelmixer=aa=0.8[logo];[0][logo]overlay=(W-
w)/2:(H-h)/2:format=auto,format=yuv420p',
        ])
        .videoCodec(codec)
    .on('start', (commandLine) => {
        console.log('FFmpeg process started:', commandLine);
    })
    .on('end', () => {
        console.log('FFmpeg process finished.');
```

```

    })
    .on('error', (err, stdout, stderr) => {
        console.error('Error:', err);
        console.error('FFmpeg stderr:', stderr);
    });
    this.ffmpeg.run();
}

killProcess(): void {
    this.ffmpeg?.kill('SIGINT');
}
}

```

processWatcher.ts

```

import si from 'systeminformation';
import path from 'path';
import jsonfile from 'jsonfile';

import { constants } from './constants';
import { VideoCodec } from './videoModuleTester';

interface MetricPacket {
  cpuUsage: number;
  ramUsage: number;
}

export default class ProcessWatcher {
  private watcherTimeout: NodeJS.Timeout | undefined;
  private metrics: MetricPacket[] = [];
  private fileName: string | undefined;
  private codec: string | undefined;
  private tool: string | undefined;

  startWatching(tool: string, task: string, resolution: string, codec: string): void {
    this.fileName = `${task}-${resolution}.json`;
    this.codec = codec;
    this.tool = tool;
    this.watcherTimeout = setInterval(async () => {
      const cpuUsage = +(await si.currentLoad()).currentLoad.toFixed(1);
      const ramUsage = +(process.memoryUsage.rss() / (1024 * 1024)).toFixed(2);
      this.metrics.push({ cpuUsage, ramUsage });
    }, 300);

    async stopWatching(): Promise<void> {
      clearInterval(this.watcherTimeout);
      let sumCpu = 0;
      let maxCpu = this.metrics[0].cpuUsage;
      let minCpu = this.metrics[0].cpuUsage;
      let sumRam = 0;
      let maxRam = this.metrics[0].ramUsage;
      let minRam = this.metrics[0].ramUsage;
      this.metrics.forEach(({ cpuUsage, ramUsage }) => {
        sumCpu += cpuUsage;
        sumRam += ramUsage;
        if (cpuUsage > maxCpu) {
          maxCpu = cpuUsage;
        }
        if (ramUsage > maxRam) {
          maxRam = ramUsage;
        }
        if (cpuUsage < minCpu) {
          minCpu = cpuUsage;
        }
        if (ramUsage < minRam) {
          minRam = ramUsage;
        }
      });
      const avgCpu = sumCpu / this.metrics.length;
      const avgRam = sumRam / this.metrics.length;
      const overallMetrics = { codec: this.codec, tool: this.tool, minCpu, maxCpu,
        avgCpu, minRam, maxRam, avgRam };
      this.metrics = [];
      this.fileName && await jsonfile.writeFile(path.join(process.cwd(),
        constants.STATISTICS_FOLDER, this.fileName), overallMetrics, { flag: 'a', EOL: ',' });
    }
  }
}

```

helpers.ts

```

import { Resolution, VideoCodec } from './videoModuleTester';
import { constants } from './constants';

export function getFileName(task: string, tool: string, codec?: VideoCodec,

```

```

resolution?: string): string {
    return `${task}-${tool}-${codec ? codec : VideoCodec.H265}-${resolution ?
resolution : constants.INPUT_RESOLUTION}`;
}
`

```

index.ts

```

import Pino from 'pino';

import SimpleFfmpeg from './simpleFfmpeg';
import FluentFfmpeg from './fluentFfmpeg';
import Gstreamer from './gstreamer';
import { Resolution, Task, VideoCodec, VideoModuleTester } from './videoModuleTester';
import ProcessWatcher from './processWatcher';
import { constants } from './constants';

const logger = Pino();

function main(task: Task): void {
    const simpleFfmpeg = new SimpleFfmpeg();
    const fluentFfmpeg = new FluentFfmpeg();
    const gStreamer = new Gstreamer();
    const tools: VideoModuleTester[] = [simpleFfmpeg, fluentFfmpeg, gStreamer];

    const processWatcher = new ProcessWatcher();

    let timeoutMultiplier = 0;
    for (const tool of tools) {
        for (const key in VideoCodec) {
            setTimeout(() => {
                logger.info(`Starting ${task} process with ${tool.getName()}, codec
${key}`);
                processWatcher.startWatching(tool.getName(), task,
constants.INPUT_RESOLUTION, key);
                switch (task) {
                    case Task.TRANSCODE:
                        tool.transcode(key, constants.INPUT_RESOLUTION);
                        break;
                    case Task.RESIZE:
                        tool.resize(key, Resolution.Low240p);
                        break;
                    case Task.SET_FPS:
                        tool.setFps(key, 15);
                        break;
                    case Task.WATERMARK:
                        tool.addWatermark(key);
                }
                setTimeout(async () => {
                    tool.killProcess();
                    await processWatcher.stopWatching();
                }, 25 * 1000);
            }, 30 * 1000 * (timeoutMultiplier));
            timeoutMultiplier += 1;
        }
    }
}

main(Task.TRANSCODE);
setTimeout(() => {
    main(Task.RESIZE);
}, 11 * 60 * 1000);
setTimeout(() => {
    main(Task.SET_FPS);
}, 22 * 60 * 1000);
setTimeout(() => {
    main(Task.WATERMARK);
}, 33 * 60 * 1000);

```