

**Politechnika Łódzka**  
**Wydział Fizyki Technicznej, Informatyki  
i Matematyki Stosowanej**

***Marcin Mazur***  
**242467**

PRACA DYPLOMOWA  
inżynierska  
na kierunku Informatyka Stosowana

**Wykorzystanie oprogramowania Open-Source do  
współpracy z kamerami TP-Link TAPO**

Instytut Informatyki I72

**Promotor:** dr inż. Krzysztof Lichy

ŁÓDŹ 2026



# **Spis treści**

<b>Streszczenie</b>	<b>3</b>
Słowa kluczowe . . . . .	3
<b>Wstęp</b>	<b>4</b>
<b>Cel i zakres pracy</b>	<b>5</b>
<b>1 Wprowadzenie technologiczne kamer IP</b>	<b>7</b>
1.1 Zastosowanie kamer IP . . . . .	7
1.1.1 Monitoring . . . . .	8
1.1.2 Kontrola dostępu . . . . .	8
1.1.3 Zarządzanie procesami biznesowymi . . . . .	9
1.1.4 Technologie Smart . . . . .	9
1.1.5 Analiza danych . . . . .	9
1.2 Budowa . . . . .	10
1.2.1 Budowa fizyczna - hardware . . . . .	10
1.2.2 Oprogramowanie - firmware . . . . .	11
1.3 Zasada działania . . . . .	11
1.3.1 Komunikacja sieciowa . . . . .	11
1.3.2 Przetwarzanie sygnału audiowizualnego . . . . .	15
1.3.3 Transmisja danych w czasie rzeczywistym . . . . .	19
1.4 Funkcje . . . . .	22
1.4.1 Obrót PTZ . . . . .	23
1.4.2 Wykrywanie obiektów i zdarzeń . . . . .	23
1.4.3 Wykrywanie ruchu . . . . .	24
1.4.4 Dwukierunkowe audio . . . . .	25
1.4.5 Zapis danych . . . . .	25
1.4.6 Zdalny dostęp i sterowanie . . . . .	26
1.4.7 Powiadomienia push . . . . .	26
1.5 Ograniczenia . . . . .	27
1.5.1 Ograniczenia wynikające z infrastruktury sieciowej . . . . .	27
1.5.2 Luki w zabezpieczeniach i ryzyka dla prywatności . . . . .	29
1.5.3 Ograniczenia modelu biznesowego i uzależnienie od producenta	32
1.5.4 Zjawisko ograniczonego potencjału sprzętowego . . . . .	32
1.6 Analiza kamery TP-Link Tapo C200 . . . . .	33
1.6.1 Charakterystyka ogólna . . . . .	33
1.6.2 Architektura sprzętowa . . . . .	34
1.6.3 Architektura oprogramowania i protokoły komunikacyjne . . . . .	36

1.6.4	Analiza możliwości funkcjonalnych . . . . .	38
1.6.5	Ograniczenia i zjawisko „vendor lock-in” . . . . .	40
1.6.6	Aspekty bezpieczeństwa i prywatności . . . . .	40
1.7	Wnioski i analiza . . . . .	43
<b>2</b>	<b>Metodologia i implementacja rozwiązania</b>	<b>45</b>
2.1	Metodyka projektowa . . . . .	45
2.1.1	Double Diamond . . . . .	45
2.2	Architektura rozwiązania . . . . .	47
2.2.1	Architektura wielowarstwowa . . . . .	48
2.2.2	Wzorzec architektury potokowej . . . . .	52
2.2.3	Wzorzec architektury opartej na zdarzeniach . . . . .	53
2.3	Diagramy . . . . .	54
2.3.1	Diagram architektury systemu . . . . .	54
2.3.2	Diagram klas . . . . .	55
2.3.3	Diagram sekwencji - Proces nagrywania . . . . .	57
2.4	Zastosowane narzędzia i technologie . . . . .	58
2.4.1	Język programowania . . . . .	58
2.4.2	Zarządzanie zależnościami . . . . .	59
2.4.3	Ekosystem konteneryzacji . . . . .	60
2.4.4	Interfejs webowy i protokół komunikacji . . . . .	60
2.4.5	Biblioteki przetwarzania multimediiów . . . . .	61
2.4.6	Kontrola kamery . . . . .	63
2.4.7	Narzędzie do kompozycji i zapisu danych . . . . .	64
2.5	Proces implementacji rozwiązania . . . . .	65
2.5.1	Provisioning i pierwotna konfiguracja środowiska kamery . . . . .	65
2.5.2	Konfiguracja środowiska programistycznego . . . . .	66
2.5.3	Implementacja serwera HTTP . . . . .	68
2.5.4	Implementacja warstwy dostępu do sprzętu . . . . .	68
2.5.5	Abstrakcja sterowania mechaniką (PTZ) . . . . .	69
2.5.6	Realizacja strumieniowania wideo . . . . .	70
2.5.7	Realizacja strumieniowania audio . . . . .	72
2.5.8	Implementacja warstwy komunikacyjnej . . . . .	74
2.5.9	Budowa interfejsu użytkownika . . . . .	76
2.5.10	Detekcja zdarzeń . . . . .	77
2.5.11	Moduł rejestracji . . . . .	80
2.5.12	Archiwizacja . . . . .	81
2.5.13	Konteneryzacja i wdrożenie . . . . .	82
2.6	Podsumowanie . . . . .	83

<b>3 Testowanie i analiza wyników</b>	<b>85</b>
3.1 Środowisko testowe . . . . .	85
3.2 Przebieg scenariuszy testowych . . . . .	86
3.2.1 Test T01: Analiza wydajności algorytmu detekcji ruchu . . . . .	86
3.2.2 Test T02: Stabilność i płynność strumieniowania wideo . . . . .	87
3.2.3 Test T03: Wpływ procesu rejestracji na zasoby . . . . .	89
3.3 Wnioski i analiza . . . . .	90
3.3.1 Algorytm detekcji ruchu . . . . .	90
3.3.2 Stabilność i płynność strumieniowania . . . . .	91
3.3.3 Analiza wpływu procesu rejestracji na zasoby . . . . .	93
3.4 Synteza wniosków . . . . .	95
<b>Wnioski Końcowe</b>	<b>97</b>
<b>Kierunki dalszego rozwoju</b>	<b>99</b>
<b>Spis rysunków</b>	<b>103</b>
<b>Spis tabel</b>	<b>104</b>

## **Streszczenie**

Niniejsza praca inżynierska podejmuje problematykę ograniczonego potencjału sprzętowego oraz uzależnienia od infrastruktury chmurowej (zjawisko vendor lock-in) w konsumenckich kamerach IoT, na przykładzie modelu TP-Link Tapo C200.

Głównym celem pracy było zaprojektowanie i implementacja autorskiego, modułowego oprogramowania typu Open Source, pozwalającego na pełną obsługę kamery w sieci lokalnej, z pominięciem dedykowanej aplikacji producenta. Projekt zrealizowano zgodnie z metodyką Double Diamond, a rozwiązanie wdrożono w języku Python 3.13 z wykorzystaniem konteneryzacji Docker.

W warstwie technologicznej zastosowano bibliotekę OpenCV do przetwarzania obrazu i autorskiej detekcji ruchu, PyAV do obsługi ścieżki audio oraz bibliotekę PyTapo do sterowania mechaniką PTZ (Pan-Tilt-Zoom) poprzez inżynierię wsteczną protokołów producenta. Interfejs użytkownika zbudowano w oparciu o serwer Flask i protokół WebSocket, co pozwoliło na komunikację i podgląd w czasie rzeczywistym.

Przeprowadzone testy wydajnościowe potwierdziły osiągnięcie stabilnego strumieniowania wideo (średnio 15,7 FPS) oraz skuteczność algorytmu detekcji ruchu działającego na brzegu sieci. Zidentyfikowano również ograniczenia związane z narzutem środowiska Python oraz zarządzaniem pamięcią RAM podczas rejestracji nagrań. Finalny produkt stanowi funkcjonalną bramę IoT.

## **Słowa kluczowe**

IoT, Kamera IP, TP-Link Tapo, Open Source, PyTapo, Docker, RTSP, Detekcja Ruchu, Flask.

## **Wstęp**

Globalny rynek systemów monitoringu przechodzi dynamiczną transformację, będącą efektem rozwoju **Internetu Rzeczy**. Kamery IP stały się wszechobecnym elementem infrastruktury cyfrowej, pełniąc funkcje od podstawowego dozoru, aż po zaawansowaną analizę danych. Równolegle z postępem technologicznym, pojawia się wyzwanie o charakterze inżynierskim, jakim jest występowanie systemów opartych na **zamkniętym oprogramowaniu**.

Wybór tematu pracy wynika z konieczności zaadresowania problemu **ograniczonego potencjału sprzętowego** w kontekście popularnej serii kamer konsumenckich TP-Link Tapo. Zjawisko to, polegające na uzależnieniu pełnej funkcjonalności sprzętu od infrastruktury chmurowej i aplikacji mobilnej producenta, ogranicza dostępność danych na poszczególnych platformach i stanowi ograniczenia nagrywania i późniejszego wykorzystania sprzętu do lokalnych zastosowań.

W pracy zastosowano **metodykę Double Diamond**, dzieląc proces projektowy na fazy eksploracji i definiowania problemu (analiza protokołów kamery) oraz fazy rozwoju i dostarczania rozwiązania. Warstwa aplikacyjna została zaimplementowana w języku **Python 3.13** z wykorzystaniem **konteneryzacji Docker** dla zapewnienia izolacji i wysokiej **reprodukwalności środowiska**. Komunikacja z kamerą odbywa się poprzez bibliotekę **PyTapo**, natomiast przetwarzanie strumienia wideo RTSP realizują narzędzia **PyAV** i **OpenCV**. Taki zestaw narzędzi, osadzony w architekturze serwera **Flask** z protokołem **WebSocket's**, pozwolił na stworzenie systemu o niskim opóźnieniu i wysokiej niezawodności.

# Cel i zakres pracy

## Cel

**Celem głównym** niniejszej pracy inżynierskiej jest opracowanie oraz implementacja kompletnego, modułowego rozwiązania programistycznego opartego na **otwartym oprogramowaniu (Open Source)**, które umożliwi pełne wykorzystanie funkcjonalności kamery IP TP-Link Tapo C200 w środowisku lokalnym i uniezależni użytkownika od zamkniętej infrastruktury producenta.

Osiągnięcie celu głównego jest weryfikowane poprzez realizację następujących, **konkretnych i mierzalnych** celów szczegółowych:

- Umożliwienie stabilnego **wyświetlania obrazu w czasie rzeczywistym**. Weryfikacja nastąpi poprzez pomiar **opóźnienia strumienia wideo** oraz wskaźnika **klatek na sekundę (FPS)**, celem osiągnięcia płynności monitoringu.
- Sterowanie kluczowymi funkcjami kamery, w tym **ruchem PTZ** (Pan/Tilt/Zoom).
- Implementacja **algorytmu wykrywania ruchu**, z poziomu serwera hostującego. Weryfikacja nastąpi poprzez analizę **efektywności algorytmów** mierzoną w kategoriach czasu przetwarzania klatki oraz minimalizacji błędów detekcji.
- Zbudowanie rozwiązania w oparciu o technologię **Docker** w celu zapewnienia **skalowalności systemu** oraz **reprodukwalności środowiska** na platformach mikserwerowych IoT (np. Raspberry Pi).
- Implementacja funkcjonalności **zapisu nagrani wideo** na serwerze hostującym z możliwością ich późniejszego **odtwarzania** poprzez interfejs webowy.

## Zakres Pracy

Zakres pracy inżynierskiej obejmuje projektowanie, implementację oraz testowanie modułowego systemu klient-serwer. Praca stanowi odpowiedź na problem *ograniczonego potencjału sprzętowego w segmencie kamer IoT*, uzasadniając wybór tematu rosnącą potrzebą na otwarte systemy zarządzania danymi.

## Aspekty objęte zakresem pracy

- Projekt trójwarstwowej architektury kontenerowej dla warstwy dostępu do sprzętu, logiki biznesowej oraz warstwy prezentacji.
- Praca skupia się na przechwytywaniu jednokierunkowego strumienia wideo i audio.
- Przeprowadzenie **testów wydajnościowych** skupiających się na **zużyciu zasobów** hosta podczas ciągłej analizy strumienia wideo.

## Wyłączenia z zakresu pracy

W celu zachowania osiągalności i weryfikowalności celów w ramach pracy inżynierskiej, poniższe aspekty zostały wykluczone, ze względu na ich złożoność badawczą lub techniczną:

- Implementacja protokołu inicjalizacji(Provisioning) została wyłączona z zakresu pracy inżynierskiej. Protokół inicjalizacji kamer Tapo jest nieudokumentowany, szyfrowany i opiera się na wymianie kluczy sesji, za pośrednictwem chmury TP-Link. W konsekwencji, praca zakłada, że **kamera została jednorazowo skonfigurowana w sieci Wi-Fi** przy użyciu oficjalnej aplikacji mobilnej.
- Implementacja modeli **uczenia maszynowego** (np. rozpoznawanie twarzy, klasyfikacja obiektów), ze względu na wysokie wymagania obliczeniowe i złożoność czasową, **została wykluczona**.

# 1 Wprowadzenie technologiczne kamer IP

Rozdział ten ma za zadanie ugruntować wiedzę o proporcji złożoności systemów kamer IP. Współczesne systemy monitoringu wizyjnego oparte na kamerach IP stanowią kluczowy element infrastruktury bezpieczeństwa, wykraczając funkcjonalnością poza tradycyjne, analogowe systemy CCTV. Ewolucja ta jest ściśle związana z rozwojem sieci komputerowych i koncepcji IoT, gdzie urządzenia periferyjne uzyskują zdolność do przetwarzania i autonomicznej komunikacji w ramach sieci. Z inżynierskiego punktu widzenia, kamera IP jest zaawansowanym systemem wbudowanym, łączącym optykę, cyfrowe przetwarzanie sygnału, kompresję danych oraz kompleksowy stos protokołów sieciowych.

## 1.1 Zastosowanie kamer IP

Tabela 1.1: Główne obszary zastosowań kamer IP w różnych sektorach przemysłu i usług.

<b>Obszar zastosowania</b>	<b>Przykłady wykorzystania kamer IP</b>
Bezpieczeństwo publiczne	Monitorowanie ulic, placów, obiektów strategicznych; automatyczne wykrywanie zagrożeń i incydentów.
Transport i logistyka	Monitoring lotnisk, dworców, portów; analiza przepływu pasażerów; automatyczne rozpoznawanie tablic rejestracyjnych.
Przemysł	Kontrola procesów produkcyjnych, wykrywanie awarii maszyn, nadzór nad pracownikami i bezpieczeństwem pracy.
Handel detaliczny	Zapobieganie kradzieżom, analiza zachowań klientów, optymalizacja układu sklepu.
Edukacja	Zwiększenie bezpieczeństwa uczniów i nauczycieli, kontrola dostępu do budynków szkolnych.
Ochrona zdrowia	Nadzór nad pacjentami i personelem, zabezpieczenie pomieszczeń szpitalnych, kontrola dostępu do stref wrażliwych.
Smart City	Analiza ruchu drogowego, inteligentne sterowanie sygnalizacją świetlną, planowanie urbanistyczne na podstawie danych z kamer.

Kamery IP znalazły szerokie zastosowanie w różnych sektorach przemysłu i usług, stanowiąc kluczowy element infrastruktury bezpieczeństwa i zarządzania.

### 1.1.1 Monitoring

Podstawowym i historycznym zastosowaniem kamery IP jest **nadzór wizyjny (monitoring)**. W odróżnieniu od analogowego CCTV, monitoring oparty na protokole internetymum umożliwia przesyłanie strumienia wideo wysokiej rozdzielczości (np. 1080p) oraz metadanych poprzez standardowe sieci LAN/WLAN. Z technicznego punktu widzenia, monitoring realizowany jest poprzez ciągłe kodowanie wideo, strumieniowanie za pomocą protokołów czasu rzeczywistego oraz zapis cyfrowy na nośnikach lokalnych (np. karty microSD), serwerach NVR(Network Video Recorder) lub w chmurze.



Rysunek 1.1: Monitoring wizyjny z wykorzystaniem kamer IP w infrastrukturze miejskiej. Źródło: Moxter, b.d.

### 1.1.2 Kontrola dostępu

Kamery IP są coraz częściej integrowane z systemami **Kontroli Dostępu (Access Control Systems - ACS)**. Ich rola wykracza poza zwykłe weryfikowanie tożsamości. Kamery stają się kluczowym sensorem w bezdotykowej autoryzacji. Przykłady zastosowań inżynierskich obejmują:

- Rozpoznawanie Twarz**: Zastosowanie algorytmów głębokiego uczenia(Deep Learning) do identyfikacji i weryfikacji osób uprawnionych, automatycznie odblokowując wejścia.
- Rozpoznawanie Tablic Rejestraacyjnych**: Automatyczne zezwalanie na wjazd pojazdów do strzeżonych stref (np. parkingów pracowniczych) na podstawie analizy obrazu z kamery.

Takie rozwiązania minimalizują ryzyko błędów ludzkich i zwiększają bezpieczeństwo poprzez ciągłe logowanie zdarzeń wejścia i wyjścia, stanowiąc integralną część za-bezpieczeń fizycznych i sieciowych.

### 1.1.3 Zarządzanie procesami biznesowymi

Wykorzystanie kamer IP w zarządzaniu procesami biznesowymi koncentruje się na optymalizacji operacyjnej poprzez zbieranie danych o efektywności i bezpieczeństwie pracy. W sektorach takich jak produkcja i logistyka, kamery są używane do: **Kontroli Jakości**. Monitorowanie linii produkcyjnych w celu automatycznego wykrywania defektów, niezgodności montażu lub nieprawidłowej sekwencji działań. Analiza ścieżek ruchu pracowników i pojazdów w celu identyfikacji wąskich gardeł w magazynach i centrach dystrybucyjnych. Te zastosowania wymagają wysokiej precyzji metadanych i niskiego opóźnienia, co stawia wysokie wymagania przed **algorytmami analizy brzegowej**, które muszą działać na poziomie procesora kamery lub serwera lokalnego.

### 1.1.4 Technologie Smart

Kamery IP są fundamentalnym elementem **ekosystemów Smart Home i Smart City**. W tych kontekstach, kamera pełni rolę czujnika behawioralnego, dostarczając danych do zautomatyzowanych systemów decyzyjnych. W budownictwie intelligentnym, kamery Tapo, podobnie jak inne urządzenia IoT, jest zintegrowana za pomocą protokołów API z platformami takimi jak **Google Assistant i Amazon Alexa**. Pozwalają one np. na automatyzację zdarzeniową która po wykryciu incydentu włącza lub wyłącza inne urządzenia w ekosystemie Smart Home. „An IoT-Based Multimodal Real-Time Home Control System” b.d.

### 1.1.5 Analiza danych

Kamera IP, połączona ze sztuczną inteligencją, przestaje być pasywnym urządzeniem rejestrującym, a staje się aktywnym sensorem generującym **metadane strukturalne**. W kontekście systemów Big Data, strumień wideo jest intensywnie przetwarzany, stanowiąc bazę dla analityki w czasie rzeczywistym i prognozowania zdarzeń. Efektywne wykorzystanie danych wizyjnych do celów analitycznych obejmuje trzy główne poziomy inżynierskie:

- **Ekstrakcja Danych Statystycznych:** Dotyczy pomiarów ilościowych, takich jak gęstość obiektów czy generowanie map ciepła (*heatmaps*).
- **Analiza Behawioralna i Wzorce Trendów:** Identyfikacja nietypowych sekwencji zdarzeń, które mogą sugerować incydent bezpieczeństwa (np. pozostawiony pakunek).

- **Analityka Predykcyjna:** Przewidywanie potencjalnych przyszłych zdarzeń na podstawie historycznych i bieżących metadanych. Wymaga to integracji i waliidacji danych z wielu źródeł IoT.

## 1.2 Budowa

Kamera IP nie jest monolitycznym urządzeniem, lecz złożonym systemem wbudowanym, składającym się ze ścisłe zintegrowanych komponentów sprzętowych (hardware) i dedykowanego oprogramowania (firmware), które zarządza ich pracą. Poniższe podrozdziały szczegółowo omawiają te dwie warstwy.

### 1.2.1 Budowa fizyczna - hardware

**1.2.1.1 Obiektyw** Obiektyw stanowi „oko” kamery – jest układem soczewek skupiających światło na matrycy przetwornika. To on definiuje kąt widzenia, głębię ostrości oraz stopień zniekształceń obrazu na brzegach kadru.

**1.2.1.2 Przetwornik obrazu (matryca)** Przetwornik obrazu, najczęściej typu CMOS, zamienia padające przez obiektyw fotony na sygnał elektryczny, który następnie jest digitalizowany. Jego rozdzielczość, wielkość pikseli i czułość bezpośrednio determinują szczegółowość, dynamikę i jakość rejestrowanego obrazu.

**1.2.1.3 Procesor** Procesor pełni rolę „mózgu” kamery. Jest to zazwyczaj układ System-on-a-Chip (SoC), który integruje dedykowane bloki sprzętowe wymagane do optymalnej pracy kamery. Procesor zarządza przetwarzaniem sygnału, kodowaniem strumienia, obsługą sieci i interfejsem użytkownika.

**1.2.1.4 Pamięć** Pamięć w kamerze IP pełni dwie kluczowe funkcje: przechowuje firmware urządzenia oraz buforuje lub archiwizuje nagrania wideo. Najczęściej wykorzystuje się pamięć Flash oraz gniazdo na kartę microSD, a w rozwiązaniach profesjonalnych także zewnętrzne rejestratory NVR lub zasoby chmurowe, co umożliwia długoterminowe i bezpieczne składowanie danych.

**1.2.1.5 Moduł sieciowy** Moduł sieciowy odpowiada za komunikację z infrastrukturą IP. Może przyjmować formę przewodowego interfejsu Ethernet (gniazdo RJ45, często z obsługą PoE) lub układu radiowego Wi-Fi, realizującego bezprzewodowe połączenie z routerem lub punktem dostępowym.

**1.2.1.6 Obudowa** Obudowa zapewnia ochronę mechaniczną i środowiskową dla elektroniki kamery. W zależności od klasy urządzenia może być wykonana z metalu

lub wytrzymałego tworzywa, a także spełniać normy odporności na wodę, pył i akty vandalizmu.

**1.2.1.7 Zasilanie** Układ zasilania dostarcza energię do wszystkich komponentów kamery. W systemach profesjonalnych często stosuje się zasilanie PoE, które przesyła energię i dane jednym przewodem Ethernet, natomiast w zastosowaniach domowych popularne są oddzielne zasilacze sieciowe.

## 1.2.2 Oprogramowanie - firmware

Firmware to dedykowane oprogramowanie wbudowane w pamięć Flash kamery, które pełni rolę systemu operacyjnego i warstwy aplikacyjnej. Jest ono "pomostem" między fizycznym sprzętem a funkcjonalnością dostępną dla użytkownika. Firmware realizuje kluczowe zadania, takie jak:

- **Inicjalizacja sprzętu (Bootloader):** Pierwszy program uruchamiany połączeniu zasilania, który testuje i konfiguruje wszystkie komponenty sprzętowe.
- **Zarządzanie procesami:** Alokacja zasobów CPU i pamięci dla zadań takich jak kompresja wideo (np. do formatu H.264), obsługa strumienia RTSP, czy analiza obrazu.
- **Obsługa stosu sieciowego:** Implementacja protokołów komunikacyjnych (TCP/IP, Wi-Fi, HTTP), które umożliwiają połączenie z siecią lokalną i internetem.
- **Interfejs API:** Udostępnienie (lub, jak w przypadku Tapo, ukrycie) interfejsu programistycznego, który pozwala na sterowanie kamerą.

## 1.3 Zasada działania

Niniejszy podrozdział stanowi analizę mechanizmów operacyjnych, które definiują funkcjonalność nowoczesnej kamery IP.

### 1.3.1 Komunikacja sieciowa

Komunikacja kamery IP opiera się na warstwowej architekturze opisanej w książce Tanenbaum i Wetherall, 2011, co zapewnia modularność i interoperacyjność systemu. Poniżej przedstawiono esencję technologiczną poszczególnych warstw w kontekście analizowanego rozwiązania.

**1.3.1.1 Warstwa Fizyczna** Warstwa fizyczna stanowi najniższy poziom modelu opisanego w książce i jest fundamentem działania każdej kamery sieciowej. Odpowiada za przesyłanie nieformowanych ciągów bitów poprzez medium transmisyjne. Definiuje ona specyfikacje elektryczne, mechaniczne oraz funkcjonalne interfejsów urządzenia.

W inżynierii systemów wizyjnych warstwę tę realizuje się głównie w dwóch standardach:

- **Interfejs przewodowy (Ethernet)** Najpowszechniejszym standardem w profesjonalnych instalacjach CCTV jest standard IEEE 802.3. Fizycznym medium transmisyjnym jest tu miedziana skrętka komputerowa (najczęściej kategorii 5e lub 6) zakończona wtykiem RJ45. Standard ten definiuje różne prędkości transmisji (10/100/1000 Mbps) oraz wcześniej wspomniany mechanizmy zasilania urządzeń poprzez kabel sieciowy, co upraszcza instalację i zwiększa niezawodność systemu monitoringu.
- **Interfejs bezprzewodowy (Wi-Fi)** W kamerach konsumenckich warstwa fizyczna często opiera się na standardzie IEEE 802.11 (Wi-Fi). Medium transmisyjnym są w tym przypadku fale radiowe w pasmach 2.4 GHz (zapewniające lepszą przenikalność przez przeszkody) lub 5 GHz (oferujące wyższą przepustowość). W kontekście inżynierii sieciowej, kluczowe znaczenie ma tu modulacja sygnału oraz mechanizmy zarządzania dostępem do medium (CSMA/CA), które minimalizują kolizje i optymalizują wykorzystanie pasma.

**1.3.1.2 Warstwa Łącza Danych** Warstwa łącza danych odpowiada za niezawodną wymianę informacji pomiędzy urządzeniami znajdującymi się w tym samym segmencie sieci lokalnej. Jej kluczowym zadaniem jest pakowanie danych z warstw wyższych w struktury zwane ramkami oraz zarządzanie dostępem do medium transmisyjnego. W kontekście systemów monitoringu IP, warstwa ta pełni dwie krytyczne funkcje: adresację fizyczną oraz segmentację ruchu.

**Adresacja fizyczna (MAC)** Każda kamera IP posiada unikalny adres fizyczny MAC (*Medium Access Control*), wypalony na etapie produkcji w karcie sieciowej urządzenia. Jest to 48-bitowy identyfikator, który w inżynierii systemów bezpieczeństwa wykorzystywany jest do:

1. **Identyfikacji urządzenia:** Umożliwia jednoznaczna identyfikację kamery w sieci lokalnej, co jest kluczowe dla zarządzania wieloma punktami końcowymi w systemach CCTV.
2. **Rezerwacji adresacji (DHCP Reservation):** Gwarantuje, że kamera po każdym restarcie otrzyma ten sam adres IP od routera, co jest kluczowe dla stabilności nagrywania.
3. **Filtracji dostępu (MAC Filtering):** Zabezpieczenia sieci poprzez dopuszczenie do komunikacji wyłącznie urządzeń o znanych identyfikatorach sprzętowych.

**Protokół ARP** Protokół ARP pełni funkcję "tłumacza" na styku warstwy drugiej i trzeciej, mapując logiczne adresy IP na fizyczne adresy MAC. W systemach monitoringu jest on niezbędny, aby rejestrator NVR lub komputer klienta mógł zlokalizować kamerę w sieci lokalnej i przesyłać do niej ramkę. Należy jednak zaznaczyć, że bezstanowość tego protokołu stanowi istotny wektor zagrożenia. Kamery IP są podatne na ataki typu **ARP Spoofing** (zatrutwanie tablicy ARP), w których atakujący rozsyła fałszywe pakiety ARP, podszywając się pod bramę sieciową. Pozwala to na realizację ataku *Man-in-the-Middle*, umożliwiającego przechwycenie strumienia wideo.

**1.3.1.3 Warstwa Sieciowa** Warstwa sieciowa odpowiada za adresowanie logiczne oraz wyznaczanie tras (routing) pakietów pomiędzy różnymi sieciami. To właśnie na tym poziomie, kamera przestaje być widoczna tylko jako urządzenie lokalne spięte kablem lub falą radiową, a staje się autonomicznym węzłem w strukturze protokołu internetowego. Kluczowe zadania tej warstwy w systemach wizyjnych to:

**Adresacja logiczna (IPv4)** W przeciwieństwie do niezmennego adresu fizycznego MAC, adres IP jest przydzielany programowo i określa lokalizację urządzenia w topologii sieci. W kamerach Tapo i większości systemów CCTV dominuje protokół IPv4 (32-bitowy). Z inżynierskiego punktu widzenia, krytyczny jest sposób przydzielania tego adresu:

1. **Dynamiczny:** Automatyczne przypisanie adresu IP przez serwer DHCP (zazwyczaj wbudowany w router). Jest to wygodne rozwiązanie dla użytkowników domowych, jednakże może prowadzić do zmiany adresu IP kamery po restarcie urządzenia.
2. **Statyczny:** Ręczne przypisanie adresu w konfiguracji urządzenia. Jest to **obligatoryjna praktyka** w profesjonalnych wdrożeniach, gwarantująca, że punkt końcowy API kamery będzie zawsze dostępny pod tym samym adresem.

**1.3.1.4 Warstwa Transportowa** Warstwa transportowa odpowiada za logiczne połączenie między procesami ("end-to-end") oraz zarządzanie przepływem danych. W inżynierii strumieniowania wideo jest to miejsce kluczowej decyzji projektowej: wybór między niezawodnością dostarczenia danych a minimalizacją opóźnień. Kamery IP wykorzystują równolegle dwa protokoły transportowe, obsługujące różne aspekty swojego działania.

**Protokół UDP (User Datagram Protocol)** UDP jest protokołem bezpołączeniowym, charakteryzującym się minimalnym nagłówkiem i brakiem mechanizmów potwierdzania odbioru. W kontekście kamer IP jest on domyślnym fundamentem dla transportu

właściwego strumienia wideo. Z inżynierskiego punktu widzenia, użycie UDP jest po-dyktowane specyfiką danych czasu rzeczywistego: spóźniony pakiet jest bezużyteczny. W monitoringu "na żywo" znacznie korzystniejsze jest porzucenie uszkodzonej klatki obrazu, niż zatrzymanie strumienia w celu oczekiwania na retransmisję, co generowa-łoby narastające opóźnienia i efekt "buforowania".

**Protokół TCP (Transmission Control Protocol)** TCP jest protokołem połączeno-wym, gwarantującym dostarczenie wszystkich pakietów w poprawnej kolejności. Ka-mery IP wykorzystują go w obszarach, gdzie integralność danych jest ważniejsza niż czas:

1. **Płaszczyzna sterowania:** Obsługa negocjacji sesji (RTSP handshake), przesy-łanie poleceń PTZ oraz komunikacja z API kamery. Utrata komendy "przesuń w lewo" jest niedopuszczalna, stąd wymóg użycia TCP.
2. **Aktualizacje firmware'u:** Proces ten wymaga niezawodnego transferu dużych plików binarnych, gdzie każdy błąd może skutkować uszkodzeniem oprogramo-wania układowego. TCP zapewnia mechanizmy kontroli błędów i retransmisji, co jest kluczowe dla integralności aktualizacji.

**Porty i multipleksacja usług** Warstwa czwarta wprowadza pojęcie portów, co po-zwala kamerze na jednoczesne działanie jako serwer wielu usług pod jednym adresem IP. Typowa mapa portów nowoczesnej kamery obejmuje:

- **Port 554:** Standardowy port dla strumieniowania RTSP.
- **Port 80 / 443:** Serwer HTTP/HTTPS obsługujący interfejs webowy oraz API dla aplikacji mobilnych (np. Tapo App).
- **Port 2020:** Często zarezerwowany dla protokołu integracyjnego ONVIF.

**1.3.1.5 Warstwa Aplikacji** Warstwa aplikacji jest najwyższym poziomem modelu OSI, stanowiącym bezpośredni interfejs dla użytkownika końcowego oraz zewnętrznych systemów informatycznych. To tutaj zachodzi właściwa interpretacja danych i re-alizacja konkretnych usług biznesowych. W ekosystemie kamer IP warstwa ta integruje szereg protokołów definiujących sposób, w jaki urządzenie jest wykrywane, sterowane i w jaki sposób udostępnia strumień mediów.

**Strumieniowanie i Sterowanie (RTSP/ONVIF)** Najważniejszą usługą aplikacyjną jest **RTSP (Real-Time Streaming Protocol)**. Pełni on rolę "pilota zdalnego sterowa-nia" dla sesji multimedialnej. RTSP działa w tandemie z protokołem RTP (Real-Time Transport Protocol), który odpowiada za faktyczny transport danych multimedialnych.

Oba te protokoły zostaną opisane w dalszej części pracy. Dla zapewnienia interoperacyjności kluczowy jest standard **ONVIF (Open Network Video Interface Forum)**. Jest to zbiór usług Web Services opartych na języku XML/SOAP, który standaryzuje komunikację. Dzięki niemu aplikacja napisana w Pythonie może w ten sam sposób wykonywać sterowanie obrót PTZ kamery marki TP-Link, jak i urządzenia w pełni profesjonalnego, np. Axis czy Hikvision.

**GUI i API (HTTP/HTTPS)** Protokół HTTP, stanowiący fundament Internetu, w kameras IP pełni podwójną rolę:

1. **Interfejs Webowy:** Serwuje graficzny panel konfiguracyjny (GUI) dostępny przez przeglądarkę, umożliwiając zmianę ustawień sieciowych, parametrów obrazu czy aktualizację firmware'u.
2. **API:** Służy jako nośnik dla programistycznych interfejsów sterowania. W przypadku kamer Tapo, komunikacja z aplikacją mobilną odbywa się poprzez szyfrowane zapytania HTTPS przesyłające ładunki JSON, co pozwala na bezpieczne sterowanie urządzeniem z poziomu smartfona.

Poprawne funkcjonowanie warstwy aplikacji zależy od szeregu usług tła:

- **DHCP (Dynamic Host Configuration Protocol):** Automatyzuje proces konfiguracji sieciowej, pobierając adres IP, maskę i bramę od routera tuż po uruchomieniu kamery.
- **NTP (Network Time Protocol):** Jest krytyczny dla materiału dowodowego. Synchronizuje wewnętrzny zegar kamery z globalnym wzorcem czasu. Brak synchronizacji (np. data z roku 1970) może sprawić, że nagranie z włamania zostanie odrzucone przez sąd jako niewiarygodne.
- **DNS (Domain Name System):** Umożliwia kamerze komunikację z chmurą producenta przy użyciu nazw domenowych zamiast trudnych do zapamiętania i zmiennych adresów IP.

### 1.3.2 Przetwarzanie sygnału audiowizualnego

Sercem kamery IP jest jej zdolność do przekształcania zjawisk fizycznych – światła i dźwięku – w ustrukturyzowany, skompresowany strumień danych cyfrowych, gotowy do transmisji przez sieć. Proces ten nie jest prostą konwersją, lecz złożonym, wieloetapowym potokiem przetwarzania, realizowanym w czasie rzeczywistym przez wyspecjalizowane komponenty sprzętowe wewnątrz układu System-on-a-Chip (SoC). Architektura SoC jest tu kluczowa; zamiast obciążać uniwersalny procesor zadaniami intensywnymi obliczeniowo, deleguje je do dedykowanych, wysoce wydajnych bloków sprzętowych. Dzięki temu kamera działa nie jak tradycyjny komputer, ale jak wyspecjalizowana

"rafineria danych", której jedynym celem jest nieustanne przekształcanie ogromnego strumienia surowych danych sensorycznych w zoptymalizowany, użyteczny produkt końcowy – skompresowany strumień audiowizualny.

**1.3.2.1 Ścieżka Przetwarzania Obrazu** Droga, jaką przebywa informacja wizualna od obiektywu do interfejsu sieciowego, jest najbardziej złożonym procesem wewnątrz kamery.

**1. Akwizycja w Matrycy CMOS** Wszystko zaczyna się w przetworniku obrazu, którym w nowoczesnych kamerach jest niemal wyłącznie matryca CMOS (Complementary Metal-Oxide-Semiconductor).

- **Konwersja fotonów na ładunek:** Gdy światło przechodzi przez obiektyw, fotony uderzają w siatkę milionów światłoczułych elementów na matrycy, zwanych fotodiodami. Każda fotodioda, pod wpływem energii fotonów, generuje ładunek elektryczny, którego wielkość jest wprost proporcjonalna do intensywności padającego na nią światła.
- **Filtr Bayera:** Fotodiody same w sobie są "ślepe" na kolory – mierzą jedynie luminację (natężenie światła). Aby uzyskać informację o kolorze, powierzchnia matrycy jest pokryta mozaiką mikroskopijnych filtrów w trzech podstawowych kolorach: czerwonym (R), zielonym (G) i niebieskim (B). Najczęściej stosowany jest tzw. filtr Bayera, w którym na każdy kwadrat 2x2 piksele przypadają dwa filtry zielone, jeden czerwony i jeden niebieski. Wynika to z faktu, że ludzkie oko jest najbardziej wrażliwe na światło zielone. W rezultacie, na wyjściu z matrycy otrzymujemy surowy, "mozaikowy" obraz, w którym każdy piksel zawiera информацию tylko o jednym z trzech kolorów.
- **Odczyt i digitalizacja:** W przeciwieństwie do starszych matryc CCD, w technologii CMOS każda fotodioda (lub mała grupa) ma swój własny, zintegrowany wzmacniacz i obwody odczytu. Pozwala to na szybki, bezpośredni odczyt wartości ładunku z każdego piksela i jego konwersję na sygnał cyfrowy jeszcze na poziomie samego sensora lub w jego bezpośrednim sąsiedztwie.

**2. Przetwarzanie w ISP (Image Signal Processor)** Surowy, zdigitalizowany obraz w formacie Bayera jest następnie przekazywany do dedykowanego koprocesora – Procesora Sygnału Obrazu (ISP). ISP to potężny, wyspecjalizowany układ, często będący częścią głównego SoC, który w czasie rzeczywistym wykonuje serię skomplikowanych operacji w celu przekształcenia surowych danych w pełnowartościowy, estetyczny obraz wideo. Potok przetwarzania w ISP (ISP Pipeline) obejmuje następujące kluczowe etapy:

Tabela 1.2: Etapy przetwarzania w potoku ISP.

Etap	Opis
<b>Akwizycja Danych Surowych (Bayer)</b>	Otrzymanie zdigitalizowanego, mozaikowego obrazu z matrycy CMOS, gdzie każdy piksel reprezentuje natężenie tylko jednego z trzech kolorów ( <i>R, G lub B</i> ).
<b>Interpolacja Kolorów</b>	Algorytm rekonstruuje pełną informację o kolorze ( <i>RGB</i> ) dla każdego piksela poprzez interpolację brakujących wartości na podstawie kolorów sąsiednich pikseli.
<b>Redukcja Szumów</b>	Zastosowanie zaawansowanych filtrów w celu usunięcia szumu cyfrowego, który powstaje zwłaszcza przy słabym oświetleniu ( <i>wysokie ISO</i> ).
<b>Automatyczna Korekcja (AWB/AE)</b>	Analiza całej sceny w celu automatycznego dostosowania <b>balansu bieli</b> ( <i>AWB</i> ) dla naturalnego odzworowania kolorów oraz <b>ekspozycji</b> ( <i>AE</i> ) dla optymalnej jasności obrazu.
<b>Ulepszanie Obrazu</b>	Zastosowanie operacji takich jak korekcja gamma, regulacja kontrastu, nasycenia kolorów oraz wyostrzanie krawędzi w celu poprawy ogólnej jakości wizualnej.
<b>Konwersja Przestrzeni Kolorów</b>	Przekształcenie obrazu z przestrzeni kolorów <i>RGB</i> na format bardziej odpowiedni do kompresji wideo, najczęściej <b>YCbCr</b> , który oddziela informację o jasności ( <i>Y</i> ) od informacji o kolorze ( <i>Cb, Cr</i> ).

Po przejściu przez potok ISP, mamy do czynienia z pełnokolorowym, skorygowanym, ale wciąż nieskompresowanym strumieniem wideo. Strumień ten, nawet dla rozdzielczości 1080p przy 30 klatkach na sekundę, ma ogromną przepływność (rzędu 1.5 Gb/s), co czyni go niemożliwym do przesłania przez typową sieć domową.

**3. Kompresja Wideo (H.264/H.265)** Ostatnim etapem przetwarzania obrazu jest jego drastyczna kompresja. Przetworzony, nieskompresowany strumień wideo (w formacie YCbCr) jest kierowany do kolejnego wyspecjalizowanego bloku sprzętowego w SoC – sprzętowego kodera wideo. W nowoczesnych kamerach są to kodery implementujące standardy H.264 lub H.265. International Telecommunication Union, 2019a; International Telecommunication Union, 2019b.

- **Zasada działania:** Kodery te wykorzystują zaawansowane techniki w celu redukcji redundancji przestrzennej (wewnątrz pojedynczej klatki) i temporalnej (pośród kolejnymi klatkami). Analizują obraz w poszukiwaniu podobnych bloków i zamiast przesyłać pełną informację o każdym z nich, przesyłają tylko informacje o różnicach i wektorach ruchu.
- **Sprzęt vs. Oprogramowanie:** Realizacja kompresji H.264 w czasie rzeczywistym jest zadaniem niezwykle wymagającym obliczeniowo. Próba wykonania jej programowo na głównym CPU kamery byłaby zbyt wolna i energochłonna. Dlatego kluczowe jest użycie dedykowanego bloku sprzętowego, który wykonuje te operacje wielokrotnie szybciej i przy znacznie niższym zużyciu energii.

**1.3.2.2 Ścieżka Przetwarzania Dźwięku** Proces przetwarzania dźwięku jest mniej złożony niż obrazu, ale podąża za podobną logiką konwersji i kompresji.

**1. Akwizycja w Mikrofonie MEMS** Dźwięk jest przechwytywany przez mikrofon wykonany w technologii MEMS (Micro-Electro-Mechanical Systems). Fale dźwiękowe wprawiają w drgania miniaturową membranę wewnątrz mikrofonu. W najpopularniejszych mikrofonach pojemnościowych, te drgania zmieniają pojemność elektryczną, co jest przekształcane na analogowy sygnał elektryczny.

**2. Digitalizacja i Konwersja PDM do PCM** Współczesne mikrofony MEMS są urządzeniami wysoce zintegrowanymi i często zawierają w swojej obudowie przetwornik analogowo-cyfrowy (ADC).

- **Modulacja Sigma-Delta:** ADC w mikrofonie to zazwyczaj modulator sigma-delta, który z bardzo wysoką częstotliwością (rzędu kilku MHz) próbuje przybliżyć wartość sygnału analogowego, generując na wyjściu jednabitowy strumień danych zwany PDM (Pulse Density Modulation). Gęstość impulsów w tym strumieniu odpowiada amplitudzie oryginalnego sygnału audio.
- **Konwersja do PCM:** Strumień PDM jest następnie przesyłany do głównego układu SoC. Tam, dedykowany blok cyfrowego przetwarzania sygnałów (DSP) stosuje filtr dolnoprzepustowy aby usunąć szum kwantyzacji przeniesiony na wysokie częstotliwości przez modulator i proces decymacji czyli zmniejszenia częstotliwości próbkowania. W rezultacie jednabitowy strumień PDM o wysokiej częstotliwości jest konwertowany na standardowy, wielobitowy strumień PCM (Pulse Code Modulation) o typowej częstotliwości próbkowania dla audio (np. 8, 16 lub 44.1 kHz). PCM to nieskompresowana, cyfrowa reprezentacja dźwięku.

**3. Kompresja Audio (AAC)** Podobnie jak w przypadku wideo, surowy strumień audio PCM ma zbyt dużą przepływność do efektywnej transmisji. Jest on więc kierowany do kodera audio, który kompresuje go przy użyciu stratnego kodeka, najczęściej AAC (Advanced Audio Coding).

- **Kodowanie percepcyjne:** AAC wykorzystuje model psychoakustyczny do analizy dźwięku i usuwania tych jego składowych, które są niesłyszalne lub maskowane przez inne, głośniejsze dźwięki dla ludzkiego ucha. Pozwala to na znaczną redukcję rozmiaru danych przy minimalnej odczuwalnej utracie jakości. AAC jest standardem w wielu zastosowaniach strumieniowych, w tym na platformach takich jak YouTube czy w urządzeniach Apple.

**1.3.2.3 Synchronizacja i Muksowanie** Ostatnim krokiem wewnętrz SoC, zanim dane trafią do karty sieciowej, jest połączenie oddzielnych, skompresowanych strumieni wideo (H.264) i audio (AAC) w jeden spójny strumień. Proces ten, zwany multipleksowaniem (muksowaniem), polega na przeplataniu pakietów audio i wideo w ramach jednego kontenera. Kluczowe jest przy tym osadzenie w strumieniu precyzyjnych znaczników czasu dla każdego pakietu, co pozwoli aplikacji klienckiej na idealne zsynchronizowanie odtwarzania obrazu i dźwięku. Po tym etapie, gotowy, zsynchronizowany strumień danych jest przekazywany do interfejsu sieciowego w celu opakowania go w pakiety RTP i wysłania w sieć.

### 1.3.3 Transmisja danych w czasie rzeczywistym

Po przetworzeniu i skompresowaniu danych audiowizualnych, ostatnim zadaniem kamery jest ich efektywna transmisja do klienta przez sieć. Proces ten, znany jako strumienianie (streaming), opiera się na zestawie wyspecjalizowanych protokołów warstwy aplikacji, które zarządzają sesją i transportują dane w sposób zoptymalizowany pod kątem czasu rzeczywistego.

**1.3.3.1 Separacja Sterowania i Danych: Rola RTSP i RTP** Fundamentalną zasadą architektoniczną w strumieniowaniu na żywo jest rozdzielenie płaszczyzny sterowania od płaszczyzny danych. Oznacza to, że protokół używany do zarządzania sesją, np. uruchamiania i zatrzymywania strumienia, jest inny niż protokół używany do faktycznego przesyłania pakietów z wideo i audio. To rozdzielenie pozwala na zoptymalizację każdego z tych zadań z osobna: sterowanie wymaga niezawodności, a przesyłanie danych – szybkości i niskich opóźnień. W przypadku kamer IP, najczęściej stosowanym zestawem protokołów do strumieniowania jest kombinacja **RTSP (Real-Time Streaming Protocol)** i **RTP (Real-time Transport Protocol)**.

1. **RTSP (Real-Time Streaming Protocol):** Pełni rolę "sieciowego pilota zdalnego sterowania". Schulzrinne, Rao i Lanphier, 1998. Jest to protokół warstwy aplikacji, który służy do nawiązywania, kontrolowania i kończenia sesji strumieniowej. Klient używa komend RTSP, aby "powiedzieć" kamerze, co ma robić – np. "zaczni j nadawać", "zatrzymaj na chwilę" czy "zakończ transmisję". Ponieważ utrata polecenia sterującego byłaby problematyczna, komunikacja RTSP odbywa się zazwyczaj za pośrednictwem niezawodnego protokołu TCP. Co istotne, RTSP nie transportuje samych danych multimedialnych – jego zadaniem jest jedynie zarządzanie sesją.
2. **RTP (Real-time Transport Protocol):** Jest to protokół odpowiedzialny za transport danych. Schulzrinne i in., 2003. Jego zadaniem jest opakowanie skompresowanych danych wideo i audio w pakiety RTP i przesłanie ich do klienta. Aby zminimalizować opóźnienia, RTP niemal zawsze działa na bazie szybkiego protokołu UDP. Każdy pakiet RTP zawiera informacje niezbędne do prawidłowego odtworzenia strumienia po stronie klienta, takie jak numery sekwencyjne i znaczniki czasu.

**1.3.3.2 Nawiązywanie Sesji Strumieniowej: RTSP Handshake** Zanim na ekranie klienta pojawi się pierwszy obraz, musi on przeprowadzić z kamerą negocjacje za pomocą protokołu RTSP. Ten proces, często nazywany "uściskiem dloni" (handshake), przebiega w kilku krokach i jest niezbędny do ustalenia parametrów transmisji. Klient wysyła do kamery serię komend RTSP, a kamera odpowiada odpowiednimi komunikatami statusu. Podstawowe komendy używane w tym procesie to:

Tabela 1.3: Podstawowe komendy protokołu RTSP.

Komenda	Nadawca	Cel
<b>DESCRIBE</b>	Klient	Żądanie od serwera (kamery) opisu dostępnych strumieni multimedialnych. Odpowiedź zawiera dane w formacie <b>SDP</b> , informujące np. o istnieniu strumienia video H.264 i audio AAC.
<b>SETUP</b>	Klient	Konfiguracja transportu dla każdego strumienia z osobna. Klient informuje serwer, na których portach <b>UDP</b> będzie nasłuchiwał na pakiety <i>RTP</i> (dane) i <i>RTCP</i> (dane kontrolne).
<b>PLAY</b>	Klient	Polecenie dla serwera, aby rozpoczął transmisję pakietów <i>RTP</i> na wcześniej uzgodnione porty.
<b>PAUSE</b>	Klient	Wstrzymanie transmisji strumienia bez zrywania sesji. Sesję można wznowić komendą <b>PLAY</b> .
<b>TEARDOWN</b>	Klient	Zakończenie sesji strumieniowej i zwolnienie zasobów po stronie serwera.

Przebieg negocjacji:

1. **DESCRIBE:** Klient wysyła do kamery żądanie DESCRIBE, pytając o zawartość dostępną pod danym adresem RTSP (np. `rtsp://192.168.1.100/stream1`). Kamера odpowiada, wysyłając opis w formacie SDP (Session Description Protocol), który informuje klienta, że dostępne są dwa strumienie: jeden video (zakodowany w H.264) i jeden audio (zakodowany w AAC).
2. **SETUP:** Klient, chcąc odbierać oba strumienie, wysyła dwa osobne żądania SETUP – jedno dla strumienia video i jedno dla audio. W każdym żądaniu SETUP

klient podaje kamerze numery portów, na których będzie nasłuchiwał na przychodzące pakiety RTP (z danymi) oraz RTCP (z informacjami kontrolnymi).

3. **PLAY:** Po pomyślnym skonfigurowaniu obu strumieni, klient wysyła jedno polecenie PLAY. Jest to sygnał dla kamery, aby rozpoczęła wysyłanie pakietów RTP z danymi wideo i audio na porty wskazane przez klienta w krokach SETUP. Od tego momentu rozpoczyna się właściwe strumieniowanie.

**1.3.3.3 Transport Danych z Użyciem RTP** Gdy sesja jest już ustanowiona, kamera zaczyna wysyłać ciągły strumień pakietów RTP. Struktura tych pakietów jest kluczowa dla prawidłowego odtworzenia mediów po stronie klienta. Najważniejsze pola w nagłówku RTP to:

- **Payload Type (Typ Ładunku):** 7-bitowe pole, które identyfikuje format danych w pakiecie. Dzięki niemu klient wie, czy dany pakiet zawiera dane wideo H.264, audio AAC, czy inny typ mediów. Pozwala to na skierowanie pakietu do odpowiedniego dekodera.
- **Sequence Number (Numer Sekwencyjny):** 16-bitowy licznik, który jest inkrementowany o jeden dla każdego wysłanego pakietu RTP. To pole jest absolutnie krytyczne. Pozwala klientowi wykryć utratę pakietów (jeśli w sekwencji pojawi się luka) oraz przywrócić prawidłową kolejność pakietów, które mogły dotrzeć do celu w złej kolejności z powodu różnych dróg w sieci.
- **Timestamp (Znacznik Czasu):** 32-bitowe pole, które odzwierciedla moment próbkowania danych zawartych w pakiecie. Jest ono generowane na podstawie wewnętrznego zegara kamery. Znaczniki czasu są niezbędne do synchronizacji różnych strumieni (np. wideo i audio), do obliczania i kompensowania opóźnień sieciowych (tzw. jitter) oraz do zapewnienia płynnego odtwarzania.

## 1.4 Funkcje

Współczesna kamera IP jest czymś znacznie więcej niż pasywnym rejestratorem obrazu. Ewolucja technologiczna przekształciła ją w aktywne, wielofunkcyjne urządzenie sensoryczne, którego możliwości wykraczają daleko poza tradycyjny monitoring. Zdolność do zdalnego sterowania, intelligentnej analizy obrazu i dźwięku, działania w trudnych warunkach oświetleniowych oraz integracji z szerszymi ekosystemami cyfrowymi definiuje jej nowoczesną tożsamość. Niniejszy podrozdział stanowi przegląd kluczowych funkcji, które decydują o wszechstronności tych urządzeń.

### 1.4.1 Obrót PTZ

Funkcjonalność PTZ (Pan-Tilt-Zoom) jest jedną z najbardziej charakterystycznych cech, która odróżnia kamery dynamiczne od statycznych. Jest to zdolność do mechanicznego sterowania polem widzenia kamery w trzech osiach, co znaczowo rozszerza jej możliwości operacyjne.

- **Pan (Obrót poziomy):** Odnosi się do ruchu kamery w płaszczyźnie poziomej, od lewej do prawej, co pozwala na skanowanie szerokich panoram.
- **Tilt (Pochylenie pionowe):** Oznacza ruch w płaszczyźnie pionowej, w górę i w dół, umożliwiając obserwację obiektów na różnych wysokościach.
- **Zoom (Powiększenie):** Zdolność do zmiany ogniskowej obiektywu w celu przybliżenia lub oddalenia obrazu. Należy rozróżnić dwa typy zoomu:
  - **Zoom optyczny:** Realizowany przez fizyczny ruch soczewek w obiektywie. Zmienia on powiększenie bez utraty jakości obrazu, co jest kluczowe dla identyfikacji szczegółów z dużej odległości, takich jak twarze czy tablice rejestracyjne.
  - **Zoom cyfrowy:** Jest to w rzeczywistości powiększenie fragmentu już przechwyconego obrazu, co prowadzi do interpolacji pikseli i nieuchronnej degradacji jakości.

### 1.4.2 Wykrywanie obiektów i zdarzeń

Integracja sztucznej inteligencji (AI) i uczenia maszynowego (ML) bezpośrednio w kamerze (tzw. Edge AI) jest jedną z najważniejszych innowacji w dziedzinie monitoringu. Dzięki potężnym procesorom wbudowanym w układy SoC, kamery zyskały zdolność do analizowania obrazu w czasie rzeczywistym, przekształcając się z pasywnych rejestratorów w inteligentne sensory.

**1.4.2.1 Detekcja i klasyfikacja obiektów** W przeciwieństwie do prostej detekcji ruchu, algorytmy AI oparte na głębokich sieciach neuronowych (np. YOLO) potrafią identyfikować i klasyfikować konkretne obiekty w polu widzenia kamery. Kamera jest w stanie odróżnić człowieka od pojazdu, zwierzęcia czy poruszającej się na wietrze gałęzi. Główną korzyścią jest drastyczna redukcja fałszywych alarmów, co pozwala operatorom skupić się na realnych zagrożeniach.

**1.4.2.2 Wykrywanie zdarzeń i analiza behawioralna** Zaawansowane modele AI idą o krok dalej, rozpoznając nie tylko obiekty, ale również ich zachowania i zdarzenia. Przykłady obejmują:

- **Przekroczenie wirtualnej linii (Line Crossing):** Wykrycie obiektu przecinającego zdefiniowaną w kadrze linię.
- **Wykrywanie wtargnięcia (Intrusion Detection):** Alarmowanie, gdy obiekt wejdzie do określonej, zabronionej strefy.
- **Wykrywanie wałesania się (Loitering Detection):** Identyfikacja osoby lub pojazdu przebywającego w danym obszarze dłużej niż ustalony czas.
- **Klasyfikacja dźwięku:** Niektóre kamery potrafią analizować również sygnał audio, rozpoznając dźwięki takie jak tłuczone szkło, krzyk czy strzał z broni palnej.

Analityka brzegowa (Edge Analytics) oznacza, że te skomplikowane obliczenia odbywają się na samej kamerze, co minimalizuje opóźnienia, zmniejsza obciążenie sieci i serwerów oraz zwiększa prywatność, ponieważ często tylko metadane (np. "wykryto osobę o godzinie 14:32") są wysyłane do chmury, a nie cały strumień wideo.

#### 1.4.3 Wykrywanie ruchu

Jest to bardziej podstawowa, ale wciąż fundamentalna funkcja, dostępna w niemal każdej kamerze IP. Jej celem jest identyfikacja jakiejkolwiek zmiany w obserwowanej scenie, która może wskazywać na ruch. W przeciwieństwie do detekcji obiektów opartej na AI, tradycyjne metody detekcji ruchu są prostsze obliczeniowo i nie "rozumieją", co jest źródłem ruchu. Najczęściej stosowane są dwie techniki:

- **Różnica międzyklatkowa (Frame Differencing):** Algorytm ten porównuje kolejne klatki wideo piksel po pikselu. Jeśli różnica w wartościach pikseli w określonym obszarze przekroczy zdefiniowany próg, system uznaje to za ruch. Jest to metoda bardzo szybka, ale podatna na fałszywe alarmy spowodowane np. zmianami oświetlenia.
- **Odejmowanie tła (Background Subtraction):** Ta bardziej zaawansowana technika polega na stworzeniu statystycznego modelu tła (tego, jak scena wygląda, gdy nic się w niej nie porusza). Każda nowa klatka jest porównywana z tym modelem, a znaczące różnice są klasyfikowane jako obiekty pierwszego planu, czyli ruch. Metoda ta jest bardziej odporna na globalne zmiany oświetlenia, ale może być mylona przez powolne zmiany w tle lub poruszające się obiekty, które są jego częścią (np. falujące na wietrze drzewa).

Wykrycie ruchu jest najczęściej wykorzystywane jako wyzwalacz (trigger) dla innych akcji, takich jak rozpoczęcie nagrywania na karcie SD lub wysłanie powiadomienia push do użytkownika.

#### **1.4.4 Dwukierunkowe audio**

Funkcja dwukierunkowego audio przekształca kamerę z pasywnego urządzenia nasłuchowego w interaktywny system. Dzięki wbudowanemu mikrofonowi i głośnikowi, użytkownik może nie tylko słyszeć dźwięk z otoczenia kamery, ale również mówić przez nią, a jego głos zostanie odtworzony przez głośnik urządzenia.

Ta dwukierunkowa komunikacja jest realizowana cyfrowo, a dane audio w obie strony są przesyłane przez tę samą sieć IP, co strumień wideo. Z technicznego punktu widzenia, implementacja tej funkcji często opiera się na protokołach Voice over IP (VoIP), takich jak SIP (Session Initiation Protocol) do nawiązywania i zarządzania sesją oraz RTP (Real-time Transport Protocol) do transportu pakietów audio w czasie rzeczywistym.

Zastosowania tej funkcji są bardzo szerokie:

- **Komunikacja:** Rozmowa z domownikami, dziećmi czy zwierzętami domowymi.
- **Weryfikacja:** Rozmowa z gościem lub kurierem stojącym przed drzwiami.
- **Odstraszanie:** Możliwość verbalnego ostrzeżenia potencjalnego intruza, co często jest skutecznym środkiem prewencyjnym.

#### **1.4.5 Zapis danych**

Kamery IP oferują kilka elastycznych metod zapisu i archiwizacji materiału wideo, co pozwala dostosować rozwiązanie do konkretnych potrzeb w zakresie bezpieczeństwa, budżetu i infrastruktury sieciowej.

**1.4.5.1 Zapis lokalny na karcie microSD** Wiele kamer, zwłaszcza z segmentu konsumenckiego, jest wyposażonych w gniazdo na kartę pamięci microSD. Umożliwia to zapis nagrani bezpośrednio na urządzeniu, bez potrzeby korzystania z zewnętrznych rejestratorów czy połączenia z internetem. Jest to rozwiązanie idealne do zapisu zdarzeń wyzwalanych ruchem w lokalizacjach o ograniczonej łączności sieciowej. Główną wadą jest ryzyko utraty nagrani w przypadku kradzieży lub fizycznego uszkodzenia samej kamery.

**1.4.5.2 Rejestrator sieciowy (NVR)** Network Video Recorder (NVR) to dedykowane urządzenie w sieci lokalnej, którego zadaniem jest odbieranie strumieni wideo z wielu kamer IP i zapisywanie ich na wbudowanych dyskach twardych. NVR stanowi centralny punkt zarządzania systemem monitoringu, oferując dużą pojemność zapisu, możliwość ciągłego nagrywania 24/7 oraz zaawansowane funkcje odtwarzania i wyszukiwania. Jest to standardowe rozwiązanie w profesjonalnych systemach bezpieczeństwa.

**1.4.5.3 Zapis w chmurze (Cloud Storage)** W tym modelu strumień wideo z kamery jest przesyłany przez internet i zapisywany na serwerach dostawcy usługi. Główne zalety to:

- **Zdalny dostęp:** Nagrania są dostępne z dowolnego miejsca na świecie za pośrednictwem aplikacji mobilnej lub przeglądarki internetowej.
- **Bezpieczeństwo danych:** Materiał jest bezpieczny nawet w przypadku kradzieży lub zniszczenia kamery.
- **Brak lokalnego sprzętu:** Eliminuje potrzebę zakupu i utrzymania NVR.

Wadą tego rozwiązania jest uzależnienie od stałego połączenia z internetem, miesięczne koszty subskrypcji oraz potencjalne obawy dotyczące prywatności danych.

#### 1.4.6 Zdalny dostęp i sterowanie

Jedną z kluczowych cech kamer IP jest możliwość zdalnego dostępu i sterowania nimi za pośrednictwem sieci komputerowej, w tym internetu. Ta funkcjonalność umożliwia użytkownikom monitorowanie i zarządzanie swoimi kamerami z dowolnego miejsca na świecie, korzystając z różnych urządzeń, takich jak smartfony, tablety czy komputery. Współczesne kamery IP oferują kilka metod zdalnego dostępu:

- **Aplikacje mobilne:** Producenci kamer często dostarczają dedykowane aplikacje na systemy iOS i Android, które umożliwiają łatwe połączenie z kamerą pochodzącej z producenta. Aplikacje te oferują funkcje takie jak podgląd na żywo, odtwarzanie nagrani, konfiguracja ustawień kamery oraz otrzymywanie powiadomień o zdarzeniach.
- **Przeglądarki internetowe:** Wiele kamer IP posiada wbudowany serwer WWW, który pozwala na dostęp do interfejsu zarządzania kamery za pomocą standardowej przeglądarki internetowej. Użytkownicy mogą logować się do kamery, przeglądać strumień wideo i zmieniać ustawienia bez konieczności instalowania dodatkowego oprogramowania.

#### 1.4.7 Powiadomienia push

Powiadomienia push to mechanizm natychmiastowego informowania użytkownika o zdarzeniach wykrytych przez kamerę, bez konieczności ciągłego obserwowania obrazu na żywo. Architektura tego systemu opiera się na współpracy kilku elementów:

1. **Wykrycie zdarzenia:** Kamera wykrywa zdarzenie, takie jak ruch, dźwięk lub detekcja obiektu przez AI.
2. **Komunikacja z serwerem:** Kamera (lub jej oprogramowanie) wysyła informację o zdarzeniu do serwera producenta w chmurze.

3. **Wysłanie do bramki push:** Serwer producenta kontaktuje się z dedykowaną bramką powiadomień dla danego systemu operacyjnego – APNs (Apple Push Notification service) dla urządzeń z systemem iOS lub FCM (Firebase Cloud Messaging) dla urządzeń z systemem Android.
4. **Dostarczenie do urządzenia:** Bramka APNs/FCM dostarcza powiadomienie na odpowiednie urządzenie mobilne użytkownika.
5. **Wyświetlenie alertu:** System operacyjny telefonu wyświetla powiadomienie na ekranie, często wraz z krótkim opisem i zrzutem ekranu ze zdarzenia, co pozwala użytkownikowi na natychmiastową reakcję.

Ten mechanizm, oparty na modelu publikacji i subskrypcji, jest niezwykle wydajny i oszczędny dla baterii urządzenia mobilnego, ponieważ nie wymaga stałego połączenia aplikacji z serwerem.

## 1.5 Ograniczenia

Pomimo dynamicznego rozwoju i szerokiego spektrum zastosowań, technologia kamer IP obarczona jest szeregiem fundamentalnych ograniczeń. Wynikają one zarówno z natury samej technologii, jak i z modeli biznesowych przyjętych przez producentów sprzętu. Pełne zrozumienie tych ograniczeń jest kluczowe dla projektowania świadomych inżyniersko, niezależnych i bezpiecznych systemów monitoringu. Niniejszy podrozdział dokonuje systematycznej analizy tych wyzwań, grupując je w trzy wzajemnie powiązane domeny: ograniczenia wynikające z infrastruktury sieciowej, luki w zabezpieczeniach i ryzyka dla prywatności oraz ograniczenia narzucone przez ekosystem producenta.

### 1.5.1 Ograniczenia wynikające z infrastruktury sieciowej

Podstawową cechą definiującą kamerę IP jest jej funkcjonowanie jako węzła w sieci komputerowej. Ta fundamentalna zależność sprawia, że wydajność i niezawodność kamery są nierozerwalnie związane z jakością i przepustowością infrastruktury sieciowej, w której operuje. Ograniczenia te są szczególnie dotkliwe w typowych wdrożeniach konsumenckich, gdzie sieć domowa rzadko jest optymalizowana pod kątem ciągłej transmisji wideo w czasie rzeczywistym.

**1.5.1.1 Wymagania Dotyczące Przepustowości i Zużycie Danych** Transmisja strumienia wideo, zwłaszcza w wysokiej rozdzielczości, jest procesem wysoce zasobochłonnym, który generuje stałe i znaczące obciążenie dla sieci. Wielkość tego obciążenia nie jest stałą wartością, lecz dynamiczną funkcją czterech kluczowych zmiennych:

- **Rozdzielczość:** Wyższa rozdzielczość oznacza większą liczbę pikseli w każdej klatce, co przekłada się na bardziej szczegółowy obraz, ale jednocześnie wykładowiczo zwiększa ilość danych do przesłania. Strumień wideo w rozdzielczości 1080p (Full HD) wymaga zazwyczaj przepustowości na poziomie 2-4 Mbps, podczas gdy strumień 4K (Ultra HD) może z łatwością konsumować od 8 do 15 Mbps, a nawet więcej.
- **Liczba klatek na sekundę:** Parametr ten definiuje płynność ruchu w nagraniu. Zwiększenie liczby klatek z 15 do 30 FPS podwaja ilość przesyłanych danych, co bezpośrednio przekłada się na proporcjonalny wzrost wymaganego pasma. Redukcja FPS jest skuteczną metodą ograniczenia zużycia pasma, jednak odbywa się kosztem utraty płynności, co może być krytyczne przy analizie szybkich zdarzeń.
- **Kompresja:** Wybór kodeka ma fundamentalne znaczenie dla efektywności transmisji. Nowocześniejszy standard H.265 (HEVC) jest w stanie zredukować wymagania dotyczące przepustowości nawet o 50% w porównaniu do powszechnie stosowanego H.264, przy zachowaniu porównywalnej jakości wizualnej.
- **Złożoność sceny:** Nowoczesne kodeki wideo optymalizują transmisję, kodując głównie zmiany pomiędzy kolejnymi klatkami. W rezultacie, statyczna scena, taka jak pusty korytarz, będzie generować znacznie mniejszy strumień danych niż dynamiczna scena z dużą ilością ruchu, np. wejście do sklepu w godzinach szczytu. Wysoka aktywność w kadrze może nawet podwoić chwilowe zapotrzebowanie na pasmo.

**1.5.1.2 Zależność od Stabilności i Jakości Połączenia Sieciowego** Protokół transmisji w czasie rzeczywistym (RTP), stanowiący podstawę strumieniowania wideo z kamer IP, jest zoptymalizowany pod kątem minimalizacji opóźnień, a nie gwarancji dostarczenia danych. W praktyce oznacza to, że w przypadku utraty pakietu danych w sieci, nie jest on retransmitowany, aby nie powodować zatrzymania ("zacięcia") obrazu. Ta cecha architektoniczna sprawia, że jakość strumienia jest niezwykle wrażliwa na wszelkie niedoskonałości sieci, które zdarzają się w środowiskach bezprzewodowych.

- **Utrata pakietów (Packet Loss):** Każdy utracony pakiet to bezpowrotnie utracony fragment informacji o obrazie lub dźwięku. Skutkuje to bezpośrednio widocznymi i słyszalnymi artefaktami: pikselozą (obraz staje się "kwadratowy"), zamrożeniem klatek (stuttering), zniekształconym lub przerywanym dźwiękiem, a także desynchronizacją obrazu i dźwięku. Badania wskazują, że poziom utraty pakietów na poziomie zaledwie 2% może już poważnie zdegradować jakość rozmowy wideo lub transmisji na żywo.

- **Niestabilność sieci Wi-Fi:** Zdecydowana większość kamer konsumenckich jest instalowana w sieciach Wi-Fi, które z natury są medium współdzielonym i podatnym na zakłócenia. Na jakość połączenia negatywnie wpływają:
  - **Zakłócenia:** Sygnały z sąsiednich sieci Wi-Fi, urządzeń Bluetooth, kuchennek mikrofalowych i innych urządzeń działających w zatłoczonym paśmie 2.4 GHz mogą powodować kolizje i utratę pakietów.
  - **Tłumienie sygnału:** Fizyczne przeszkody, takie jak ściany, stropy i meble, osłabiają sygnał Wi-Fi. Im dalej kamera znajduje się od routera, tym słabsze połączenie, niższa przepustowość i większe prawdopodobieństwo utraty pakietów.
  - **Konkurowanie:** Kamera musi konkurować o dostęp do pasma z każdym innym urządzeniem w sieci domowej (komputerami, smartfonami, telewizorami). W godzinach szczytowego obciążenia, gdy wiele urządzeń aktywnie korzysta z internetu, sieć staje się przeciążona, co prowadzi do opóźnień i odrzucania pakietów.
- **Opóźnienia (Latency) i Zmienna Opóźnień (Jitter):** Opóźnienie to czas potrzebny na dotarcie pakietu od kamery do odbiorcy, a jitter to miara nieregularności tych opóźnień. Nawet jeśli pakiety nie są gubione, ale docierają w nierównych odstępach czasu, może to zakłócić płynność odtwarzania. Odbiorca (np. aplikacja w telefonie) posiada bufor kompensujący niewielki jitter, ale jego przepełnienie w wyniku dużych wahań opóźnień skutkuje zacinaniem się obrazu, podczas gdy odtwarzacz czeka na spóźnione pakiety.

W praktyce, te czynniki degradujące jakość nie działają w sposób addytywny, lecz mnożnikowy. Kamera o wysokiej rozdzielczości (generująca duży strumień danych), umieszczona w dużej odległości od routera (słaby sygnał) w zatłoczonej sieci Wi-Fi (wysokie zakłócenia i utrata pakietów), doświadczy katastrofalnego spadku jakości transmisji. To właśnie ten efekt wzmacniający wyjaśnia, dlaczego doświadczenia użytkowników z kamerami IP bywają tak niespójne i trudne do zdiagnozowania – postrzegany problem często jest wynikiem nałożenia się kilku pozornie niewielkich niedoskonałości sieciowych. Paradoksalnie, główna zaleta marketingowa kamer konsumenckich – łatwość instalacji dzięki łączności bezprzewodowej – stoi w bezpośredniej sprzeczności z ich technicznym wymaganiem posiadania stabilnej, wysokoprzepustowej i niskoprzetłoczonej sieci.

### 1.5.2 Luki w zabezpieczeniach i ryzyka dla prywatności

Jako permanentnie podłączone do sieci, często instalowane i zapominane urządzenia peryferyjne, kamery IP stanowią istotny i unikalny wektor zagrożeń. Ich umiejscowie-

nie w wrażliwych, prywatnych przestrzeniach sprawia, że konsekwencje udanego ataku wykraczają daleko poza typowe incydenty bezpieczeństwa IT, bezpośrednio naruszając prywatność i fizyczne bezpieczeństwo użytkowników.

**1.5.2.1 Wektory Ataków i Powszechnie Podatności** Połączenie niezabezpieczonych konfiguracji domyślnych, zaniedbań ze strony użytkowników oraz dużej powierzchni ataku czyni kamery IP głównym celem masowych, zautomatyzowanych ataków.

- **Słabe lub domyślne poświadczenia:** Głównym i najprostszym wektorem ataku jest niezmienienie przez użytkownika fabrycznych, domyślnych danych logowania (nazwy użytkownika i hasła). Atakujący wykorzystują zautomatyzowane skany, które przeszukują internet w poszukiwaniu urządzeń odpowiadających na standardowych portach, a następnie próbują uzyskać do nich dostęp, używając publicznie znanych, domyślnych poświadczeń dla danego modelu kamery Open Web Application Security Project, 2018.
- **Ekspozycja w sieci publicznej:** Błędna konfiguracja routera, w szczególności niepotrzebne przekierowanie portów, może wystawić interfejs administracyjny kamery bezpośrednio na publiczny internet. Takie urządzenia stają się łatwo wykrywalne za pomocą wyspecjalizowanych wyszukiwarek, takich jak Shodan, które indeksują podłączone do internetu urządzenia Neshenko i in., 2019.
- **Wykorzystanie w botnetach:** Przejęte kamery, ze względu na ich liczbę i stałe podłączenie do sieci, są cennym zasobem do tworzenia botnetów. Złowrogi przykład botnetu Mirai pokazał, jak setki tysięcy skompromitowanych urządzeń IoT, w dużej mierze kamer IP, zostały wykorzystane do przeprowadzenia zmasowanych ataków typu DDoS (Distributed Denial of Service), które zakłóciły działanie największych serwisów internetowych Antonakakis i in., 2017. Incydent ten unaocznił, jak indywidualne zaniedbanie bezpieczeństwa może przyczynić się do globalnej destabilizacji internetu.

**1.5.2.2 Ryzyka Związane z Oprogramowaniem Firmware** Firmware, czyli oprogramowanie układowe, pełni rolę systemu operacyjnego kamery i stanowi krytyczną, choć często niewidoczną dla użytkownika, granicę bezpieczeństwa. Powoduje to podatność na wektor ataku i dużą procentową zależność od producenta. Oto kluczowe ograniczenia związane z firmware:

- **Zamknięty kod:** W przeciwieństwie do oprogramowania open-source, firmware większości kamer konsumenckich to "czarna skrzynka". Użytkownicy i niezależni badacze bezpieczeństwa nie mają możliwości łatwego audytu kodu w poszukiwaniu luk, tylnych furtek (backdoorów) czy niebezpiecznych praktyk, takich jak zaszyte na stałe w kodzie hasła (hardcoded credentials).

- **Brak terminowych aktualizacji:** Producenci często z opóźnieniem publikują łatki bezpieczeństwa dla nowo odkrytych podatności (oznaczonych numerami CVE), a wielu użytkowników nie instaluje dostępnych aktualizacji. Stwarza to szerokie "okno możliwości" dla atakujących, którzy mogą wykorzystywać dobrze znane i opisane luki w zabezpieczeniach.
- **Polityka End-of-Life (EOL):** Jest to krytyczne, niemożliwe do obejścia ograniczenie. W momencie, gdy producent ogłasza, że dany model produktu osiągnął status EOL (koniec życia), zaprzesta wszelkiego wsparcia, w tym wydawania jakichkolwiek aktualizacji bezpieczeństwa. Każda podatność odkryta po tej dacie staje się permanentnym zagrożeniem typu "zero-day", na które nigdy nie powstanie oficjalna łatka. Biorąc pod uwagę długi cykl życia fizycznego kamer, prowadzi to do powstawania w sieci rosnącej populacji przestarzałych urządzeń, które są tykającymi bombami zegarowymi z punktu widzenia bezpieczeństwa.

**1.5.2.3 Implikacje dla Prywatności Użytkownika** Umiejscowienie kamer IP w najbardziej prywatnych przestrzeniach – domach, sypialniach, biurach – sprawia, że naruszenie bezpieczeństwa jest jednocześnie głębokim naruszeniem prywatności. Co więcej, model operacyjny oparty na usługach chmurowych wprowadza dodatkowe ryzyka związane z zarządzaniem i ochroną danych.

- **Nieautoryzowana inwigilacja:** Najbardziej bezpośrednim i dotkliwym ryzykiem jest uzyskanie przez atakującego dostępu do transmisji wideo i audio na żywo. Umożliwia to podglądanie i podsłuchiwanie domowników, co prowadziło do przypadków nękania, szantażu, a nawet szpiegostwa.
- **Bezpieczeństwo danych w chmurze:** W modelu, w którym nagrania wideo są przechowywane na serwerach producenta, użytkownik traci bezpośrednią kontrolę nad swoimi danymi. Musi on w pełni zaufać praktykom bezpieczeństwa stosowanym przez dostawcę usługi w celu ochrony przed włamaniem do infrastruktury chmurowej. Kwestie takie jak polityka prywatności, jurysdykcja przechowywania danych oraz prawa dostępu do nich stają się kluczowe. Udany atak na serwery producenta może skutkować jednoczesnym wyciekiem prywatnych nagrani tysięcy, a nawet milionów użytkowników.

Krajobraz zagrożeń IoT charakteryzuje się głęboką asymetrią ryzyka. Wysiłek wymagany od atakującego do masowego skompromitowania kamer jest niezwykle niski (np. zautomatyzowane skanowanie w poszukiwaniu domyślnych haseł), podczas gdy potencjalne konsekwencje dla ofiary są niezwykle wysokie (naruszenie prywatności, straty finansowe, nieświadomy udział w botnecie). Ten wysoce korzystny dla atakujących stosunek ryzyka do zysku gwarantuje, że tego typu ataki będą kontynuowane i będą rosły w skali. Co więcej, problem EOL nie jest jedynie kwestią techniczną,

ale bezpośrednią konsekwencją modelu biznesowego, który priorytyzuje sprzedaż nowego sprzętu nad wspieraniem istniejących produktów. Tworzy to zjawisko "planned obsolescence", w którym fizyczna funkcjonalność urządzenia znacznie przeżywa jego cyfrowe bezpieczeństwo. Decyzja biznesowa o zakończeniu wsparcia dla danego modelu przekłada się bezpośrednio na permanentną, niemożliwą do załatwiania lukę w zabezpieczeniach dla każdego użytkownika, który nie zdecyduje się na wymianę sprzętu.

### **1.5.3 Ograniczenia modelu biznesowego i uzależnienie od producenta**

Ostatnia kategoria ograniczeń nie wynika z samej technologii, lecz ze strategicznych decyzji producentów, mających na celu stworzenie zamkniętych, własnościowych eko-systemów. Działania te, motywowane biznesowo, w sposób fundamentalny ograniczają potencjał technologiczny urządzeń, interoperacyjność i długoterminowe bezpieczeństwo, co stanowi główną motywację niniejszej pracy.

### **1.5.4 Zjawisko ograniczonego potencjału sprzętowego**

Analiza rynku kamer konsumenckich ujawnia istotną dysproporcję między możliwościami technicznymi sprzętu a funkcjonalnością faktycznie udostępnioną użytkownikowi. To celowe ograniczenie potencjału sprzętowego realizowane jest przez producentów poprzez cztery główne mechanizmy:

- **Brak interoperacyjności:** Uzależnienie od konkretnej platformy ogranicza możliwość integracji z innymi systemami monitoringu lub oprogramowaniem do zarządzania wideo.
- **Właściwościowe API sterowania (Proprietary API):** Kluczowe funkcje operacyjne, takie jak sterowanie silnikami PTZ, są ukryte i niedostępne przez otwarte standardy (np. ONVIF). Wymusza to na użytkowniku korzystanie wyłącznie z oficjalnej aplikacji producenta.
- **Brak suwerenności danych:** Architektura systemu nie zapewnia użytkownikowi pełnej kontroli nad zapisem własnych danych. Jest on zmuszany do korzystania z karty SD (co niesie ryzyko utraty danych wraz z kradzieżą kamery) lub płatnej chmury, zamiast posiadać natywną możliwość łatwego zapisu nagrani na własnym, lokalnym serwerze.
- **Uzależnienie od infrastruktury chmurowej:** Pełna funkcjonalność sprzętu jest uzależniona od stałego dostępu do Internetu i serwerów producenta. Oznacza to, że w przypadku awarii sieci lub decyzji producenta o wyłączeniu serwerów, kamera traci większość swoich funkcji.

- **Dalsza integracja wykrywania ruchu i powiadomień:** Zaawansowane funkcje, takie jak wykrywanie twarzy czy analiza zachowań, są często realizowane wyłącznie po stronie chmury producenta. Ogranicza to możliwość lokalnej analizy danych i zwiększa ryzyko naruszenia prywatności.

Główną motywacją niniejszej pracy jest przełamanie zidentyfikowanych barier poprzez stworzenie oprogramowania Open Source, które zapewnia użytkownikowi pełną kontrolę i odblokowuje rzeczywisty potencjał zakupionego sprzętu.

## 1.6 Analiza kamery TP-Link Tapo C200

### 1.6.1 Charakterystyka ogólna

Kamera TP-Link Tapo C200 jest pozycjonowana na rynku jako flagowy przykład konsumentkiego urządzenia **Internetu Rzeczy (IoT)** w kategorii „**Smart Home**”. Jej podstawowym celem rynkowym jest dostarczenie masowemu odbiorcy niedrogiego, łatwego w obsłudze i bogatego w funkcje systemu monitoringu wewnętrznego, który jest w pełni zarządzany za pomocą aplikacji mobilnej. Strategia TP-Link polega na oferowaniu zaawansowanych możliwości sprzętowych w wysoce konkurencyjnej cenie, co ma na celu szybkie zdobycie udziału w rynku i wprowadzenie użytkowników do zamkniętego ekosystemu usług firmy.

Kluczowe funkcje reklamowane w oficjalnej specyfikacji technicznej stanowią fundament jej propozycji wartości:

- **Wysoka jakość obrazu:** Kamera oferuje natywną rozdzielcość 1080p Full HD ( $1920 \times 1080$  pikseli) przy płynnej prędkości 30 klatek na sekundę. Stanowi to standard rynkowy dla nowoczesnych systemów monitoringu, pozwalający na wyraźną identyfikację szczegółów.
- **Mechanizm Pan/Tilt:** Urządzenie jest wyposażone w zmotoryzowaną głowicę, umożliwiającą zdalny obrót w poziomie w zakresie  $360^\circ$  oraz pochylenie w pionie. Ta funkcja eliminuje martwe strefy i pozwala na monitorowanie całego pomieszczenia za pomocą jednego urządzenia.
- **Tryb nocny (Noktowizja):** Zintegrowane diody LED podczerwieni o długości fali 850 nm zapewniają widoczność w całkowitej ciemności na deklarowany dystans do 40 stóp (około 12 metrów).
- **Dwukierunkowe audio:** Wbudowany mikrofon i głośnik umożliwiają komunikację w czasie rzeczywistym, co przekształca kamerę z pasywnego sensora w interaktywny interkom.
- **Zaawansowana detekcja:** Poza standardową detekcją ruchu, Tapo C200 reklamuje funkcje oparte na sztucznej inteligencji (AI), w tym „**Detekcję Osób**” (Person Detection) oraz „**Detekcję Płaczu Dziecka**” (Baby Crying Detection).

Pełna funkcjonalność, począwszy od krytycznego procesu pierwszej konfiguracji (provisioningu), aż po dostęp do zaawansowanych funkcji detekcji i zdalnego podglądu, jest nierozerwalnie związana z autorską aplikacją mobilną Tapo oraz infrastrukturą chmurową TP-Link. Ten model stanowi centralny problem badawczy niniejszej pracy. Tytułowe „**Wykorzystanie oprogramowania Open-Source do współpracy z kamerami TP-Link TAPO**” jest bezpośrednią odpowiedzią inżynierską na wyzwanie, jakim jest obejście tych sztucznych ograniczeń. Niniejszy rozdział dokonuje systematycznej dekonstrukcji kamery Tapo C200, aby precyzyjnie zidentyfikować, które jej komponenty są otwarte i możliwe do integracji, a które zostały celowo zamknięte przez producenta w ramach strategii „**vendor lock-in**”. Analiza ta stanowi techniczne uzasadnienie dla zaprojektowania i implementacji niestandardowego oprogramowania opisanego w kolejnych rozdziałach pracy.

## 1.6.2 Architektura sprzętowa

Analiza architektury sprzętowej jest kluczowa dla zrozumienia zarówno potencjału, jak i ograniczeń kamery. Komponenty fizyczne definiują surowe możliwości urządzenia, które oprogramowanie układowe (**firmware**) następnie eksponuje – lub ukrywa – użytkownikowi.

Poniższa tabela syntetyzuje kluczowe specyfikacje sprzętowe, które stanowią bazę dla dalszej analizy oprogramowania i funkcjonalności.

Tabela 1.4: Kluczowe Specyfikacje Techniczne TP-Link Tapo C200

Kategoria	Specyfikacja
Przetwornik Obrazu	1/2.8" Progressive Scan CMOS
Obiektyw	Ogniskowa: 4 mm, Przysłona: F2.0
Noktowizja	Dioda IR LED 850 nm (zasiąg do 40 stóp / 12 m)
Rozdzielcość	1080P HD (1920 × 1080 px)
Szybkość Klatek	30 fps
Kompresja Wideo	H.264
System Audio	Wbudowany mikrofon i głośnik
Standard Wi-Fi	IEEE802.11b/g/n, 2.4 GHz
Zapis Lokalny	Gniazdo microSD (do 512 GB)

W zakresie systemów peryferyjnych, kamera wyposażona jest w zintegrowany mikrofon i głośnik, co stanowi techniczną podstawę dla funkcji dwukierunkowego audio. Interfejs sieciowy jest ograniczony wyłącznie do komunikacji bezprzewodowej w paśmie 2.4 GHz, obsługując standardy IEEE802.11b/g/n. Brak portu Ethernet oraz nieobsługiwanie pasma 5 GHz jednoznacznie pozycjonują C200 jako urządzenie klasy konsumenckiej, gdzie priorytetem jest łatwość instalacji bezprzewodowej, a nie maksymalna stabilność i przepustowość połączenia, jakiej wymagałyby zastosowania profesjonalne.

Centralną jednostką obliczeniową urządzenia jest wysoce zintegrowany układ **System-on-a-Chip (SoC)**. Chociaż oficjalna specyfikacja nie wymienia konkretnego modelu, analiza typowych architektur dla tego segmentu urządzeń wskazuje na użycie procesora integrującego wiele funkcji w jednym układzie (np. z serii Ingenic T31). Taki SoC łączy w sobie główny procesor (CPU), dedykowany procesor sygnału obrazu (ISP) odpowiedzialny za operacje takie jak demosaikowanie i redukcja szumów, oraz – co najważniejsze – sprzętowy koder wideo H.264.

Wybór takiej architektury SoC jest kluczową decyzją inżynierijną i biznesową. Z jednej strony, wysoka integracja drastycznie obniża koszty produkcji (**Bill of Materials - BOM**), co pozwala na oferowanie kamery w atrakcyjnej cenie. Z drugiej strony, taka

monolityczna architektura ma głębokie implikacje dla otwartości systemu. Oznacza to, że każda pojedyncza funkcja urządzenia – od ruchu silnikami PTZ, przez odczyt z sensora CMOS, aż po kompresję H.264 i zarządzanie interfejsem sieciowym – jest kontrolowana przez jeden, monolityczny obraz oprogramowania układowego dostarczany i podpisywany cyfrowo przez TP-Link. Ten wybór sprzętowy jest technicznym fundamentem, który umożliwia skuteczną implementację biznesowego modelu „**vendor lock-in**”, który zostanie szczegółowo omówiony w sekcji 2.5.

### 1.6.3 Architektura oprogramowania i protokoły komunikacyjne

Warstwa oprogramowania jest miejscem, w którym realizowana jest strategia producenta. To tutaj potencjał sprzętowy jest albo udostępniany poprzez otwarte standardy, albo celowo ograniczany przez zamknięte protokoły. Analiza Tapo C200 ujawnia świadome i celowe rozdzielenie tych dwóch podejść.

**Oprogramowanie Układowe (Firmware)** Urządzenie działa pod kontrolą zamkniętego oprogramowania układowego, bazującego najprawdopodobniej na zmodyfikowanej dystrybucji Linuksa, co jest powszechną praktyką w urządzeniach IoT. Ten firmware stanowi „**czarną skrzynkę**”, która zarządza całym sprzętem i udostępnia wszystkie usługi sieciowe. Aktualizacje firmware’u są dostarczane przez producenta w formie binarnych plików, które są cyfrowo podpisywane, co uniemożliwia użytkownikom instalację zmodyfikowanych wersji. Ta praktyka jest kluczowym elementem strategii zabezpieczeń TP-Link, ale jednocześnie stanowi barierę dla społeczności open-source, która chciałaby wprowadzać własne modyfikacje lub poprawki bezpieczeństwa.

**Standardowy Stos Sieciowy** Na poziomie sieciowym, kamera implementuje standardowy stos TCP/IP, aby móc funkcjonować w typowej sieci domowej. Obejmuje to podstawowe usługi, takie jak DHCP do automatycznej konfiguracji adresu IP, DNS do rozwiązywania nazw oraz NTP do synchronizacji czasu. Ponadto, kamera wykorzystuje HTTPS, co wskazuje na szyfrowaną komunikację, jednak ta komunikacja jest przeznaczona wyłącznie dla serwerów chmurowych TP-Link.

Prawdziwa analiza pod kątem integracji open-source zaczyna się od protokołów warstwy aplikacji, gdzie obserwujemy dychotomię:

**RTSP (Real-Time Streaming Protocol):** Specyfikacja techniczna potwierdza wsparcie dla RTSP. Jest to absolutnie kluczowy, otwarty i standardyzowany protokół, który pozwala na dostęp do surowego, skompresowanego strumienia wideo i audio. Dostępność strumienia RTSP jest fundamentalnym umożliwiaczem dla całego projektu niniejszej pracy.

**ONVIF (Open Network Video Interface Forum):** Specyfikacja również deklaruje zgodność z ONVIF. Jest to jednak przykład strategicznego „**open-washingu**” – marketingowego wykorzystania otwartego standardu w sposób, który sugeruje interoperacyjność, jednocześnie jej nie dostarczając. Ogranicza się ona w najlepszym razie do minimalnego zestawu funkcji (np. Profile S, co oznacza jedynie możliwość udostępniania strumienia video, co i tak jest już realizowane przez RTSP). Co najważniejsze, implementacja ta nie udostępnia kluczowej funkcjonalności sterowania PTZ.

**Rzeczywisty Mechanizm Sterowania: Właściwe API** Skoro ONVIF nie pozwala na sterowanie kamerą, powstaje pytanie, w jaki sposób realizuje to oficjalna aplikacja Tapo. Odpowiedź leży w istnieniu nieudokumentowanego, właściwowego protokołu sterowania. Aplikacja mobilna Tapo komunikuje się z kamerą w sieci lokalnej za pomocą niestandardowego, API. Analiza aplikacji wykryła zaszyfrowane lub zakodowane żądania w celu wykonania operacji takich jak ruch Pan/Tilt, włączenie trybu nocnego, czy zmiana ustawień detekcji.

Poniższa tabela podsumowuje analizę protokołów komunikacyjnych kamery.

Tabela 1.5: Analiza Protokołów Komunikacyjnych Tapo C200 pod kątem Integracji Open-Source

Protokół	Cel	Status	Dostępność	Użyteczność dla Projektu
RTSP	Dostęp do strumienia A/V	Otwarty Standard	Tak	Kluczowa. Stanowi podstawę do przechwytywania i analizy wideo ( <i>FFmpeg/OpenCV</i> ).
ONVIF	Interoperacyjność ( <i>Stream + Sterowanie</i> )	Otwarty Standard	Pozorne Tak	Znikoma. Implementacja jest okrojona i nie udostępnia sterowania PTZ. Nazywana „open-washingiem”.
API	Pełne sterowanie urządzeniem (PTZ, ustawienia)	Zamknięty / Własnościowy	Brak informacji	Kluczowa (Pośrednio). Wymaga <b>inżynierii wstępnej</b> . Projekt wykorzystuje <i>PyTapo</i> do obsługi tego API.
Protokół Chmurowy (HTTPS)	Zdalny dostęp, alerty, <i>provisioning</i>	Zamknięty / Własnościowy	Brak informacji	Brak. Jest to mechanizm, który projekt ma na celu ominąć, aby <b>uniezależnić się od producenta</b> .

#### 1.6.4 Analiza możliwości funkcjonalnych

Sekcja ta dokonuje ponownej oceny funkcji reklamowanych w sekcji 2.1, tym razem przez pryzmat inżynierski, oceniąc ich rzeczywistą dostępność dla dewelopera open-source, w przeciwieństwie do ich teoretycznej obecności w urządzeniu.

**Przetwarzanie i Strumieniowanie Wideo** Ta funkcja jest w pełni dostępna. Kamera niezawodnie dostarcza wysokiej jakości strumień H.264 (1080p przy 30 fps) poprzez otwarty protokół RTSP. Z punktu widzenia projektu, jest to solidny i wystarczający fundament. Pozwala na pobranie „surowca” (danych wideo), który następnie może być przetwarzany lokalnie przez autorskie algorytmy. Dostępność ta jest warunkiem koniecznym dla powodzenia całego projektu.

**Funkcjonalność PTZ (Pan/Tilt/Zoom)** W tym przypadku obserwujemy fundamentalne rozłączenie między możliwością sprzętową a dostępnością programową. Mechanizmy (silniki) do obrotu i pochylenia są fizycznie obecne w urządzeniu. Jednak, jak ustalono w sekcji 2.3, są one niedostępne przez jakikolwiek otwarty standard, taki jak ONVIF.

W konsekwencji, z perspektywy dewelopera open-source, kamera Tapo C200 bez dodatkowej inżynierii wstępnej jest funkcjonalnie kamerą statyczną. Dopiero zastosowanie biblioteki PyTapo „odblokowuje” tę natywną funkcję sprzętową, co jest jednym z głównych celów implementacyjnych niniejszej pracy.

**Wbudowane Funkcje AI** Najbardziej złożona sytuacja dotyczy wbudowanych funkcji „AI Detection”, takich jak wykrywanie osób i płaczu dziecka. Problem nie polega na tym, że te funkcje nie działają. Można założyć, że algorytmy uczenia maszynowego (prawdopodobnie uruchamiane na wyspecjalizowanym koprocesorze w ramach SoC) skutecznie analizują obraz i generują zdarzenia. Problem polega na niedostępności wyjścia tych algorytmów.

Model operacyjny TP-Link dla tych zdarzeń jest następujący:

1. Wbudowany algorytm AI na kamerze wykrywa zdarzenie (np. „osoba”).
2. Kamera nie emituje tego zdarzenia w sieci lokalnej (LAN) w formie otwartego komunikatu (np. przez MQTT, ONVIF Events, czy nawet prosty webhook).
3. Zamiast tego, kamera wysyła zaszyfrowany komunikat o zdarzeniu wyłącznie do serwerów chmurowych TP-Link.
4. Serwery TP-Link przetwarzają ten komunikat i wysyłają powiadomienie push do aplikacji mobilnej użytkownika.

Ten model, w którym metadane zdarzeń są „brane jako zakładnik” („**data hostage**”) przez infrastrukturę chmurową, czyni całą zaawansowaną, wbudowaną analitykę AI całkowicie bezużyteczną dla lokalnych systemów automatyki. Niemożliwe jest stworzenie w prosty sposób automatyzacji w systemie Home Assistant typu: „JEŻELI kamera Tapo wykryje osobę, TO włącz światło w korytarzu”.

Ta celowa blokada dostępu do danych o zdarzeniach ma kluczową implikację dla niniejszej pracy: zmusza ona do **reimplementacji** funkcjonalności, która już istnieje w urządzeniu. Skoro nie można odczytać zdarzenia „detekcja ruchu” z kamery, projekt musi sam pobrać surowy strumień video (przez RTSP) i przeprowadzić własną, serwerową analizę detekcji ruchu.

### **1.6.5 Ograniczenia i zjawisko „vendor lock-in”**

Syntezą analizy sprzętu, oprogramowania i funkcjonalności prowadzi do jednoznacznego wniosku: ograniczenia kamery Tapo C200 nie są wynikiem braków technicznych, lecz świadomą strategią biznesową ograniczenia potencjału sprzętowego znaną jako „**vendor lock-in**” (uzależnienie od dostawcy).

Z technicznego punktu widzenia, strategia „**vendor lock-in**” w przypadku Tapo C200 opiera się na trzech filarach:

- 1. Zamknięte API Sterowania (Proprietary Control API):** Jak omówiono w sekcji 2.3, brak otwartego standardu sterowania PTZ zmusza użytkowników do korzystania wyłącznie z oficjalnej aplikacji lub polegania na niestabilnych, reverse-engineeryjnych rozwiązańach, takich jak PyTapo.
- 2. Uchwycenie Metadanych AI (AI Metadata Capture):** Jak omówiono w sekcji 2.4, przesyłanie zdarzeń detekcji wyłącznie do chmury uniemożliwia lokalną automatyzację i wymusza na użytkowniku poleganie na infrastrukturze producenta w zakresie otrzymywania alertów.
- 3. Szyfrowany i Chmurowy Provisioning:** Jest to pierwszy i najbardziej fundamentalny zamek. Proces inicjalizacji kamery i jej podłączenia do sieci Wi-Fi (provisioning) jest nieudokumentowany, szyfrowany i wymaga obowiązkowej weryfikacji po stronie chmury TP-Link. Oznacza to, że kamery nie można nawet uruchomić w sieci lokalnej bez użycia oficjalnej aplikacji mobilnej i aktywnego połączenia z internetem. Jest to tak złożona bariera, że niniejsza praca musi ją zaakceptować jako ograniczenie: w założeniach projektu stwierdza się, że „praca zakłada, że kamera została jednorazowo skonfigurowana w sieci Wi-Fi przy użyciu oficjalnej aplikacji mobilnej”.

Wniosek z tej analizy jest jasny: „Wyzwanie Open Source” nie jest przypadkowym niedopatrzeniem inżynierów TP-Link. Jest to precyzyjnie zaprojektowany zestaw barier technicznych, których celem jest ochrona modelu biznesowego firmy. Praktyczna implementacja opisana w Rozdziale 3 niniejszej pracy jest zatem w swojej istocie aktem inżynierii obchodzenia (**bypass engineering**) tych celowo narzuconych ograniczeń.

### **1.6.6 Aspekty bezpieczeństwa i prywatności**

Ostatnia warstwa analizy dotyczy bezpieczeństwa i prywatności. Jest to najważniejszy argument przemawiający za koniecznością stworzenia otwartego, lokalnego rozwiązania. Model „**vendor lock-in**” nie tylko ogranicza funkcjonalność, ale także generuje poważne i udokumentowane zagrożenia dla użytkowników.

**Ryzyko dla Prywatności** Model operacyjny oparty na subskrypcji zapisu nagrań w chmurze zachęca do fundamentalnego kompromisu w zakresie prywatności. Wymaga on przesyłania wrażliwych danych – materiału audio i wideo z wnętrza prywatnego domu – na serwery firmy trzeciej. Taka architektura generuje trzy ryzyka:

- **Ryzyko wycieku danych:** Pomyślny atak na infrastrukturę chmurową TP-Link mógłby skutkować masowym wyciekiem prywatnych nagrań tysięcy użytkowników.
- **Ryzyko nadużycia:** Użytkownik traci suwerenność nad swoimi danymi i musi ufać, że pracownicy dostawcy lub jego podwykonawcy nie uzyskają nieautoryzowanego dostępu do jego danych.
- **Ryzyko prawne:** Dane przechowywane w chmurze podlegają jurysdykcji prawnej kraju, w którym znajdują się serwery, i mogą być przedmiotem żądań organów ścigania bez wiedzy użytkownika.

Lokalne rozwiązanie, do którego dąży niniejsza praca, całkowicie eliminuje te ryzyka, ponieważ dane nigdy nie opuszczają sieci lokalnej użytkownika.

**Zidentyfikowane Luki w Zabezpieczeniach** Zamknięty, nieaudytowalny firmware kamery Tapo C200 okazał się być podatny na krytyczne luki bezpieczeństwa. Nie jest to już teoretyczne ryzyko; jest to udokumentowany fakt.

## 1. CVE-2021-4045: Krytyczna Luka RCE

Najpoważniejszą znaną luką jest CVE-2021-4045, której przyznano ocenę 9.8 w skali CVSS Hacefresko, 2021.

- **Problem:** Luka typu „unauthenticated Remote Code Execution” (nieuwierzytelne zdalne wykonanie kodu).
- **Wektor:** Luka znajdowała się w binarnym pliku uhttpd – tym samym wbudowanym serwerze WWW, który był używany do obsługi własnościowego API sterującego.
- **Wpływ:** Serwer uhttpd działał z uprawnieniami użytkownika root (najwyższymi możliwymi). Oznacza to, że nieuwierzytelny atakujący w tej samej sieci mógł zdalnie przejąć całkowitą kontrolę nad kamerą. Mógł ją wyłączyć, podsłuchiwać, podglądać, a także – co być może najgroźniejsze – wykorzystać ją do ataku na inne urządzenia w sieci lokalnej użytkownika.
- **Zasięg:** Luka dotyczyła oprogramowania w wersji 1.1.15 i starszych.

## 2. Inne Wyniki Testów Penetracyjnych

Niezależne badania bezpieczeństwa potwierdziły istnienie wielu innych słabości:

- Badanie autorstwa Bella, Biondi et al. Bella i in., 2023, w ramach którego opracowano metodykę PETIoT, wykorzystało Tapo C200 jako studium przypadku. Zidentyfikowano w nim trzy nieznane wcześniej (zero-day) luki:
  - Możliwość awarii kamery w wyniku intensywnego skanowania portów czyli Denial of Service(DoS).
  - Przechwytywanie nieszyfrowanego strumienia wideo H.264 podczas korzystania z oprogramowania firm trzecich (np. przez RTSP).
  - Możliwość odgadnięcia, czy kamera wykryła ruch, na podstawie stałego rozmiaru szyfrowanych powiadomień, co stanowiło wyciek informacji.
- Ogólnym zagrożeniem dla wszystkich słabo zabezpieczonych urządzeń IoT, w tym kamer, jest ryzyko rekrutacji do botnetu (np. Mirai), który wykorzystuje ich moc obliczeniową do przeprowadzania zmasowanych ataków DDoS.

**Ostateczne Uzasadnienie dla Projektu** Powyższa analiza bezpieczeństwa i prywatności stanowi ostateczne i najsilniejsze uzasadnienie dla celu niniejszej pracy. Projek towane rozwiązanie open-source nie jest jedynie ćwiczeniem z inżynierii wstępnej w celu odblokowania funkcji PTZ. Jest to fundamentalna interwencja w zakresie bezpieczeństwa.

Tworząc w pełni funkcjonalny, lokalny serwer sterujący, rozwiązanie to daje użytkownikowi możliwość wykonania kluczowego kroku hardeningu: całkowitego zablokowania kamerze dostępu do Internetu na poziomie routera (firewalla).

Taka konfiguracja, niemożliwa przy korzystaniu z oficjalnej aplikacji, natychmiast:

- Rozwiązuje problem prywatności: Dane audio/video nigdy nie opuszczają sieci lokalnej.
- Neutralizuje ryzyko botnetu: Kamera nie może komunikować się z serwerami zewnętrznymi, więc nie może zostać zrekruitowana do botnetu.
- Omija podatny na ataki serwer: Użytkownik komunikuje się z bezpiecznym, autytowalnym serwerem Python hostowanym lokalnie, zamiast z zamkniętym firmwarem kamery.

Rozdział ten udowodnił, że TP-Link Tapo C200 jest idealnym studium przypadku konfliktu IoT. Stanowi on techniczne uzasadnienie, dlaczego proponowana w niniejszej pracy architektura – lokalna, oparta na otwartym oprogramowaniu i przywracająca użytkownikowi kontrolę – jest nie tylko pożądana z punktu widzenia funkcjonalności, ale wręcz konieczna z punktu widzenia prywatności i cyberbezpieczeństwa.

## 1.7 Wnioski i analiza

Przeprowadzona w niniejszym rozdziale analiza technologiczna systemów monitoringu IP, ze szczególnym uwzględnieniem ekosystemu TP-Link Tapo, pozwala na sformułowanie kluczowych **wniosków** determinujących kierunek prac inżynierskich opisanych w kolejnych częściach dyplomu. Kamera IP, będąca w istocie złożonym **systemem wbudowanym** (*SoC*) integrującym optykę, przetwarzanie sygnału i stos protokołów sieciowych, posiada potencjał wykraczający poza funkcjonalności udostępniane fabrycznie przez producenta. Jednakże, pełne wykorzystanie tego potencjału w otwartych systemach informatycznych napotyka na szereg barier technicznych i biznesowych.

Zidentyfikowano fundamentalną niezgodność standardów transmisji wideo z technologiami webowymi. Mimo że protokół **RTSP** (*Real-Time Streaming Protocol*) stanowi przemysłowy standard przesyłania mediów w kamerach IP, współczesne **przeglądarki internetowe nie posiadają natywnego wsparcia** dla tego protokołu ani dla surowych strumieni *H.264* transportowanych przez *UDP/TCP*.

Oznacza to, że bezpośrednia wizualizacja obrazu z kamery Tapo C200 w aplikacji internetowej, bez zastosowania pośredniczącej **warstwy transkodującej** (*middleware*), jest niemożliwa. Wymusza to zaprojektowanie autorskiego **potoku przetwarzania**, który w czasie rzeczywistym dokona translacji strumieni do formatów kompatybilnych ze współczesnymi technologiami.

Analiza modelu biznesowego producenta ujawniła zjawisko **Vendor Lock-in**, które sztucznie ogranicza funkcjonalność urządzenia w środowisku lokalnym. Kluczowe funkcje sprzętowe, takie jak sterowanie mechaniką **PTZ** (*Pan-Tilt-Zoom*) czy zapis nagrani, są dostępne wyłącznie poprzez **zamknięte, własnościowe API** lub infrastrukturę chmurową producenta.

Brak implementacji pełnego standardu **ONVIF** w modelach konsumenckich sprawia, że integracja z systemami zewnętrznymi wymaga **inżynierii wstępnej** i emulacji protokołów sterujących, co uzasadnia wykorzystanie bibliotek takich jak **PyTapo** w warstwie abstrakcji sprzętowej projektowanego rozwiązania.

Finalnie, wdrożenie rozwiązania opartego na **oprogramowaniu Open Source** (Python, OpenCV, Docker) stanowi kluczowy czynnik uwalniający potencjał integracyjny urządzenia. Przełamanie barier producenta przekształca zamkniętą kamerę konsumencką w **programowalny sensor IoT**.

Umożliwia to jej zastosowanie w zaawansowanych scenariuszach, takich jak:

- systemy kontroli dostępu,
- lokalna analiza danych przy użyciu sztucznej inteligencji,
- integracja z systemami *Smart Home*,

wszystko to bez narażania prywatności użytkownika na ryzyka związane z przetwarzaniem danych w chmurze publicznej.

## 2 Metodologia i implementacja rozwiązania

Poprzednie rozdział dokonał teoretycznej dekonstrukcji technologii kamer IP oraz przeprowadził szczegółową analizę studium przypadku — kamery TP-Link Tapo C200. Analiza ta zidentyfikowała kluczowy problem badawczy: fundamentalny konflikt między potencjałem sprzętowym urządzenia a ograniczeniami narzuconymi przez zamknięty ekosystem producenta (tzw. „**vendor lock-in**”).

Niniejszy rozdział przechodzi od teorii do praktyki. Stanowi on techniczną odpowiedź na zdefiniowane wyzwania. Opisany zostanie kompletny proces projektowy i wdrożeniowy – od wybranej metodyki badawczej, przez architekturę systemu, aż po szczegółowe implementacji poszczególnych komponentów. Celem jest budowa autorskiego, otwartego rozwiązania programistycznego, które realizuje cele postawione w niniejszej pracy.

### 2.1 Metodyka projektowa

#### 2.1.1 Double Diamond

Model Double Diamond (Podwójny Diament) jest ustrukturyzowaną metodyką procesową, pierwotnie sformalizowaną przez British Design Council w 2005 roku British Design Council, 2025. Stanowi ona mapę procesu projektowego, którego celem jest efektywne nawigowanie od wstępnej idei do wdrożonego rozwiązania, przy jednoczesnym zarządzaniu złożonością i niepewnością.

Metodyka ta jest fundamentalna dla współczesnego projektowania (w tym inżynierii oprogramowania, projektowania produktów i usług) i bazuje na koncepcji Design Thinking.

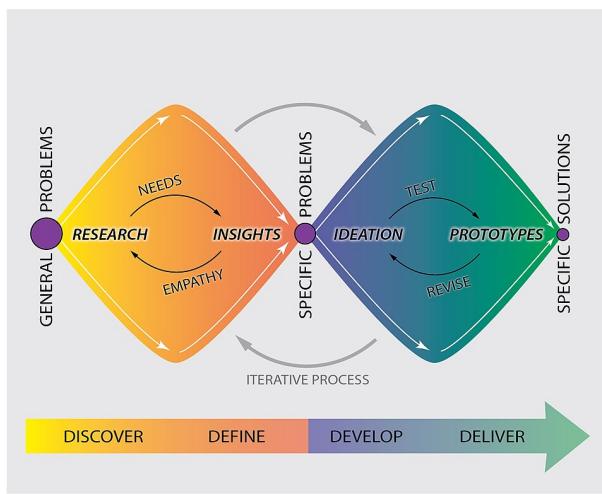
Nazwa modelu pochodzi od jego wizualnej reprezentacji jako dwóch sąsiadujących „diamentów”. Model zakłada, że aby opracować właściwe rozwiązanie, należy najpierw dogłębnie zrozumieć i zdefiniować właściwy problem. W modelu wyróżniamy:

#### Przestrzeń Problemu - Diament Pierwszy

Celem tego etapu jest zidentyfikowanie i precyzyjne zdefiniowanie kluczowego problemu, który ma zostać rozwiązany.

1. **Faza Odkrywania** - Jest to faza intensywnych badań. Projekt wychodzi poza własne założenia, aby zrozumieć rzeczywisty kontekst użytkownika i zidentyfikować jego niezaspokojone potrzeby.
2. **Faza Definiowania** – W tej fazie następuje synteza danych zebranych podczas Odkrywania. Następuje filtrowanie i analiza informacji, szukając wzorców i klu-

## DESIGN MODEL



J.A. Nykiela, University of Alberta

Rysunek 2.1: Model Double Diamond. Źródło: Design Council, 2021.

czowych wyzwań. Celem jest przekształcenie rozproszonych obserwacji w klaszowną i mierzalną definicję problemu.

### Przestrzeń Rozwiązań - Diament Drugi

3. **Faza Rozwijania** - Mając jasno zdefiniowany problem, projekt ponownie przechodzi w tryb research, aby wygenerować jak najszerzy wachlarz potencjalnych rozwiązań. Kładzie się nacisk na ilość, a nie jakość.
4. **Faza Dostarczania** - Jest to ostatnia faza, skupiona na testowaniu, walidacji i iteracyjnym udoskonalaniu wybranych koncepcji, aby ostatecznie wybrać optymalne rozwiązania gotowego do wdrożenia.

### Zastosowanie modelu w niniejszej pracy

Model Double Diamond został przyjęty jako nadzędna rama procesu projektowego w budowie lokalnej bramy czasu rzeczywistego dla kamery TP-Link Tapo C200.

1. **Odkrywanie** — Przeprowadzono badania literaturowe i technologiczne, analizując ogólną architekturę kamer IP oraz specyfikę Tapo C200. Zidentyfikowano dostępne interfejsy (RTSP, własnościowe API PTZ), ograniczenia ekosystemu producenta oraz potrzeby użytkownika: lokalna kontrola, prywatność danych, niskie opóźnienie transmisji, niezawodna detekcja ruchu i nagrywanie.
2. **Definiowanie** — Sformułowano problem główny: vendor lock-in uniemożliwia pełną, lokalną kontrolę urządzenia. Ustalono mierzalne cele i kryteria sukcesu:

stabilne strumieniowanie wideo z audio, skuteczna detekcja ruchu, rejestracja materiału, pełna kontrola PTZ bez użycia chmury, brak ekspozycji poświadczanej. Utworzono wstępny backlog wymagań oraz założenia architektoniczne.

3. **Rozwój** — Zrealizowano serię szybkich prototypów i eksperymentów technicznych: strumieniowanie i muksowanie przy użyciu PyAV, przetwarzanie klatek wideo w OpenCV, sterowanie PTZ poprzez bibliotekę PyTapo, komunikacja klient–serwer oparta o Flask i Socket.IO/WebSockets. Iteracyjnie dobierano buforowanie, wątki oraz strukturę potoków, porównując alternatywy (np. użycie gotowych platform vs. własnej aplikacji). Opracowano artefakty projektowe: diagramy architektury i przepływu.
4. **Dostarczanie** — Wdrożono docelowe rozwiązanie w aplikacji webowej, opakowane w kontener Docker dla prostoty uruchomienia. Przeprowadzono walidację: pomiary opóźnień, testy wydajności detekcji, testy zużycia pamięci oraz użytkowe próby interfejsu webowego. Wyniki posłużyły do iteracyjnych usprawnień konfiguracji strumieni, detektora ruchu i obsługi nagrywania.

## 2.2 Architektura rozwiązania

System został zaprojektowany jako **Real-Time IoT Gateway**, stanowiąca pomost między klientami internetowymi o wysokim opóźnieniu, a niskopoziomowymi protokołami sprzętowymi. U swej podstawy aplikacja opiera się na **Architekturze Trójwarstwowej**, ściśle oddzielając Warstwę Prezentacji, Logikę Aplikacji oraz Warstwę Danych. Taka separacja zapewnia, że złożoność własnościowych protokołów kamery zostaje całkowicie abstrahowana i ukryta przed interfejsem użytkownika końcowego.

W celu sprostania specyficznym wymaganiom przetwarzania audiowizualnego, system wykorzystuje hybrydę 2 wzorców architektonicznych. Pierwszym z nich jest wzorzec **Architektury Potokowej**. Zastosowany w warstwie przetwarzania do sekwencyjnej obsługi klatek wideo i fragmentów audio oraz nagrywanie i analiza ruchu.

Kolejnym wzorcem architektonicznym jest **Architektura Sterowana Zdarzeniami**, która umożliwia komunikację w czasie rzeczywistym między klientem a serwerem. Akcje użytkownika oraz zmiany stanu systemu są propagowane natychmiastowo, zamiast polegać na cyklicznym odpytywaniu (tzw. *polling*), które jest stanowczo bardziej konsumujące zasoby.

Ze względu na ciągły charakter strumieniowania wideo, architektura systemu w znacznym stopniu polega na **wielowątkowości**. Aplikacja utrzymuje współdzielony stan w pamięci operacyjnej, co pozwala na odseparowanie pętli wejściowych - odczyt ze sprzętu, od obsługi żądań wyjściowych - obsługa klientów webowych. Gwarantuje to, że obciążające procesor zadania, takie jak detekcja ruchu czy muksowanie wideo, nie blokują interfejsu użytkownika.

W kolejnych sekcjach szczegółowo omówiono odpowiedzialności poszczególnych warstw (Prezentacji, Logiki i Danych), przeanalizowano wewnętrzny przepływ danych w potokach medialnych oraz przedstawiono strukturę klas wykorzystaną do implementacji powyższej architektury.

### 2.2.1 Architektura wielowarstwowa

Współczesna inżynieria systemów **Internetu Rzeczy**, a w szczególności projektowanie bram sieciowych (*IoT Gateways*) obsługujących strumieniowanie multimedialnych w czasie rzeczywistym, wymaga rygorystycznego podejścia do strukturalizacji kodu oraz zarządzania przepływem danych. W ramach niniejszej pracy inżynierskiej, jako fundament logiczny i fizyczny rozwiązania, przyjęto **Architekturę Trójwarstwową**.

Architektura warstwowa jest powszechnie uznawana w literaturze za de facto standard w projektowaniu aplikacji korporacyjnych i systemów rozproszonych, umożliwiając dekompozycję złożonego problemu na separowalne, zarządzalne poziomy abstrakcji. Zastosowany w projekcie model trójwarstwowy dokonuje ścisłej separacji odpowiedzialności (*Separation of Concerns - SoC*) pomiędzy interakcją z użytkownikiem, logiką biznesową przetwarzania sygnału oraz fizycznym dostępem do urządzenia.

Poniższa tabela (Tabela 2.1) przedstawia szczegółowy podział odpowiedzialności w poszczególnych warstwach systemu.

Tabela 2.1: Podział warstw architektury systemu IoT

Poziom Architektury (Tier)	Rola w Systemie IoT	Odpowiedzialność Funkcjonalna
<b>Tier 1:</b> Warstwa Prezentacji	Interfejs Użytkownika (GUI), wizualizacja danych, obsługa zdań wejściowych.	Renderowanie strumienia wideo (JPEG/Canvas), panel sterowania PTZ, wyświetlanie alertów detekcji ruchu.
<b>Tier 2:</b> Warstwa Logiki	Przetwarzanie reguł, koordynacja procesów, analiza danych, transakcja protokołów.	Detekcja ruchu ( <i>Background Subtraction</i> ), obsługa sesji WebSocket, buforowanie klatek, orkiestracja wątków.
<b>Tier 3:</b> Warstwa Danych	Fizyczny dostęp do danych, abstrakcja sprzętowa, trwała pamięć masowa.	Komunikacja z API Tapo, obsługa strumienia RTSP, zarządzanie poświadczeniami.

**2.2.1.1 Warstwa Prezentacji** Warstwa prezentacji w opracowanym systemie stanowi najwyższy poziom abstrakcji, pełniący rolę interfejsu komunikacyjnego pomiędzy

użytkownikiem końcowym a logiką biznesową aplikacji. Została ona zrealizowana w formie **graficznego interfejsu użytkownika (GUI)** dostępnego z poziomu przeglądarki internetowej, co zapewnia **przenośność** i brak konieczności instalacji dedykowanego oprogramowania klienckiego.

## Zasada minimalizacji odpowiedzialności

Zgodnie z założeniami architektury trójwarstwowej, warstwa ta została zaprojektowana w sposób minimalistyczny. Jej odpowiedzialność ograniczona jest wyłącznie do dwóch funkcji:

- **Wizualizacja danych:** Prezentowanie wyników przetwarzania dostarczanych przez warstwę biznesową (obraz z kamery, statusy czujników, lista nagrani).
- **Obsługa interakcji:** Przechwytywanie akcji użytkownika (kliknięcia, sterowanie myszą) i przekazywanie ich w formie żądań do serwera.

Warstwa prezentacji **nie przetwarza obrazu ani nie zarządza połączeniem** z kamerą. Cała logika sterująca i analityczna została odseparowana i ulokowana w warstwie backendowej, minimalizując obciążenie klienta.

## Struktura i elementy po stronie klienta

Implementacja interfejsu opiera się na standardowych technologiiach webowych, dzieląc się na trzy logiczne grupy elementów, które są dostarczane do przeglądarki klienta:

- **Struktura Widoku (HTML):** Szkielet aplikacji definiujący układ elementów na ekranie. Kluczowym elementem widoku jest dedykowany obszar roboczy służący do renderowania strumienia wideo.
- **Warstwa Stylizacji (CSS):** Odpowiada za estetykę i **responsywność** interfejsu. Zapewnia czytelność paneli sterowania (PTZ) oraz adaptację układu strony do różnych rozdzielczości ekranu.
- **Logika Klientka (JavaScript):** Skrypty uruchamiane w przeglądarce, odpowiedzialne za **dynamiczną aktualizację treści** bez przeładowywania strony. Ich rola ogranicza się do nasłuchiwanego na kanały komunikacyjne i natychmiastowego odświeżania elementów w reakcji na dane napływające z serwera.

Dzięki takiemu podejściu uzyskano lekką i responsywną warstwę prezentacji, która pełni jedynie funkcję „okna” na system, delegując wszelkie obciążające zadania obliczeniowe do warstw niższych.

**2.2.1.2 Warstwa Logiki** Warstwa Logiki, umiejscowiona centralnie w architekturze trójwarstwowej, pełni rolę „systemu nerwowego” całego rozwiązania, orkiestrując przepływ danych pomiędzy użytkownikiem a sprzętem. W literaturze dotyczącej Internetu Rzeczy warstwa ta jest często definiowana jako **Middleware**, którego fundamentalnym zadaniem jest ukrycie złożoności urządzeń końcowych i udostępnienie ujednolicionych usług dla warstwy prezentacji. Działa ona jako inteligentny bufor, który transformuje surowe dane sprzętowe w użyteczne informacje biznesowe, wykorzystując mechanizmy wielowątkowości do zapewnienia płynności działania aplikacji.

### Separacja Płaszczyzn Przetwarzania

W systemach czasu rzeczywistego kluczowe jest pogodzenie obsługi ciągłych strumieni danych z interaktywnością interfejsu użytkownika, aby uniknąć zjawiska blokowania zasobów. Warstwa Logiki implementuje zaawansowany model współbieżności, dzieląc system na odseparowane płaszczyzny:

- **Płaszczyzna Danych:** Odpowiada za procesy wymagające wysokiej przepustowości i ciągłości, takie jak pobieranie i transkodowanie strumienia wideo oraz normalizacja strumienia audio. Działa ona w tle, niezależnie od aktywności użytkownika.
- **Płaszczyzna Sterowania:** Obsługuje zdarzenia inicjowane przez użytkownika, takie jak sterowanie mechaniką kamery (PTZ) czy przełączanie stanu nagrywania. Płaszczyzna ta priorytetyzuje krótki czas reakcji.
- **Płaszczyzna Komunikacji:** Realizuje warstwę transportową, wykorzystując protokoły WebSocket do komunikacji dwukierunkowej (Full-Duplex) oraz HTTP do serwowania zasobów statycznych.

Spójność między tymi płaszczyznami zapewnia **mechanizm synchronizacji stanów**. Wykorzystuje on współdzieloną pamięć operacyjną do przechowywania globalnego stanu systemu (np. flaga „trwa nagrywanie”, status „wykryto ruch”), co pozwala wątkom roboczym na natychmiastową reakcję na zmiany sterowania bez konieczności kosztownego przesyłania komunikatów międzyprocesowych.

**Płaszczyzna Danych** Urządzenia IoT, takie jak kamery monitoringu, operują na złożonych, przemysłowych standardach transmisji, które nie są natywnie wspierane przez lekkie interfejsy przeglądarkowe. Warstwa logiki działa tutaj jako „fabryka przetwarzania” w czasie rzeczywistym, dokonująca transkodowania i adaptacji sygnałów:

- **Przetwarzanie Wideo:** System odbiera surowy strumień wysokiej rozdzielczości i poddaje go procesowi skalowania oraz rekompresji. Operacja ta ma na celu

dostosowanie przepustowości strumienia do możliwości sieciowych klienta, zapewniając responsywność interfejsu nawet przy słabszym łączu internetowym.

- **Normalizacja Audio:** Warstwa ta rozwiązuje problem niekompatybilności formata dźwięku. Surowe dane z kamery są konwertowane do standardyzowanego formatu PCM. Zapobiega to powstawaniu artefaktów dźwiękowych (zniekształcenia, szумy statyczne) po stronie klienta i gwarantuje poprawną interpretację sygnału.

**Płaszczyzna Sterowania** Oprócz przetwarzania sygnałów, warstwa ta implementuje kluczowe algorytmy decyzyjne systemu:

- **Sterowanie PTZ:** Zleca wykonanie sekwencji instrukcji do sterownika silników. Logika ta uwzględnia ograniczenia fizyczne kamery (np. maksymalny kąt obrotu) oraz zapewnia płynność ruchu poprzez interpolację pozycji.
- **Zarządzanie Nagrywaniem:** Zaimplementowano maszynę stanów, która kontroluje proces rejestracji. Logika ta zarządza buforowaniem danych w pamięci RAM, synchronizacją ścieżek audio/video oraz finalną komplikacją pliku MP4, dbając o to, by operacje dyskowe zapisu nie zakłócały podglądu na żywo.

**Płaszczyzna Komunikacji** Jako punkt styku z użytkownikiem, warstwa ta pełni funkcję „dozorcy” dla systemu sprzętowego. Gdy użytkownik inicjuje akcję (np. ruch kamerą PTZ), warstwa logiki weryfikuje poprawność żądania, a następnie tłumaczy abstrakcyjną intencję na konkretną instrukcję wykonawczą dla warstwy sprzętowej. Dzięki temu separuje ona klienta webowego od fizycznych ograniczeń i specyfiki protokołów sterowania urządzeniem.

**2.2.1.3 Warstwa Danych** Najniższy poziom architektury stanowi fundament integrujący system cyfrowy ze światem fizycznym. W klasycznej inżynierii oprogramowania warstwa ta (*Data Access Layer - DAL*) odpowiada za komunikację z bazą danych. W systemach IoT pojęcie to ulega rozszerzeniu o **Warstwę Abstrakcji Sprzętu** (*Hardware Abstraction Layer - HAL*). W projekcie przyjęto założenie, że kamera IP jest specyficznym rodzajem „bazy danych”, która dostarcza strumienie informacji (video, audio) i przyjmuje polecenia modyfikacji stanu (PTZ, konfiguracja).

### **Warstwa Abstrakcji Sprzętu (HAL) jako Izolator**

Podstawowym celem implementacji HAL jest uniezależnienie wyższych warstw systemu od konkretnego modelu sprzętowego. Warstwa Logiki nie powinna operować na niskopoziomowych szczegółach, takich jak adresy URL strumieni RTSP, algorytmy szyfrowania haseł czy specyficzne kody błędów HTTP zwracane przez kamerę. Zamiast

tego, HAL udostępnia ujednolicony interfejs programistyczny (API wewnętrzne), np. metodę `camera.move_left()`, która abstrahuje złożoność implementacyjną.

W projekcie HAL realizowany jest poprzez wzorzec **Adapter**, który „opakowuje” zewnętrzne biblioteki potrzebne do akwizycji i sterowania kamerą co pozwala na:

**Łatwą wymianę sterownika:** Jeśli w przyszłości kotaś z bibliotek przestanie być rozwijana, wystarczy podmienić implementację wewnętrz klasy `TapoCamera` na inną, bez konieczności przepisywania setek linii kodu w Warstwie Logiki.

**Centralizację obsługi błędów:** HAL tłumaczy specyficzne wyjątki sieciowe czy kody błędów z kamery na zrozumiałe wyjątki domenowe, upraszczając logikę obsługi błędów w wyższych warstwach.

**Bezpieczeństwo:** HAL odpowiada za bezpieczne przechowywanie i wstrzykiwanie poświadczeń (login/hasło) do żądań. Dzięki temu dane uwierzytelniające nigdy nie „wyciekają” do warstwy prezentacji.

## 2.2.2 Wzorzec architektury potokowej

Uzupełnieniem struktury warstwowej w warstwie logiki biznesowej jest zastosowanie **Architektury Potokowej** (ang. *Pipe and Filter*). Wzorzec ten jest standardem w systemach przetwarzających strumienie danych multimedialnych, gdzie kluczowe jest zachowanie ciągłości i niskiego opóźnienia przetwarzania.

**2.2.2.1 Zastosowanie w projekcie** W zrealizowanym systemie wzorzec ten stanowi fundament działania klas zajmujących się transmisją danych audiowizualnych, które operują w nieskończonych pętlach wątków tła. Ponieważ dane z kamery napływają w sposób ciągły, każda jednostka danych (klatka wideo lub pakiet audio) musi zostać przetworzona w czasie rzeczywistym, zanim zostanie nadpisana przez kolejną. Architektura potokowa zapewnia tutaj deterministyczny przepływ danych od momentu ich akwizycji ze sprzętu aż do momentu wysłania do klienta webowego lub zapisu na dysku.

Wzorzec ten został zaimplementowany w następujących obszarach systemu:

- **Potok Wideo:** Przekształcanie surowych macierzy pikseli w obrazy JPEG wyświetlane w przeglądarce.
- **Potok Audio:** Dekodowanie, resampling i mikowanie kanałów dźwiękowych.
- **Potok Rejestracji:** Buforowanie ramek w pamięci RAM i ich finalna kompozycja do pliku MP4.

Dzięki zastosowaniu architektury potokowej, dodanie nowej funkcjonalności – np. rozpoznawania twarzy – sprowadzałoby się jedynie do wpięcia nowego „filtra” pomiędzy etap skalowania a kodowania, bez konieczności modyfikacji logiki pobierania obrazu czy komunikacji sieciowej.

**2.2.2.2 Ogólna zasada działania** Wzorzec potokowy składa się z następujących elementów:

1. **Źródło (Source):** Punkt początkowy potoku, który dostarcza surowe dane do przetworzenia. W przypadku potoku wideo jest to klatka obrazu pobrana z kamery.
2. **Zestaw filtrów (Filters):** Kolejne etapy przetwarzania danych, z których każdy wykonuje określoną transformację na danych wejściowych i przekazuje wynik do następnego filtra. Filtry są zaprojektowane jako niezależne moduły, co umożliwia ich łatwe dodawanie, usuwanie lub modyfikowanie bez wpływu na resztę potoku.
3. **Ujście (Sink):** Punkt końcowy potoku, który odbiera przetworzone dane i wykonuje na nich ostateczną operację, taką jak wysłanie do klienta lub zapis na dysku.

### **2.2.3 Wzorzec architektury opartej na zdarzeniach**

Trzecim filarem architektonicznym omawianego systemu, odpowiedzialnym za interaktywność i komunikację między warstwami, jest **Architektura Oparta na Zdarzeniach** (ang. *Event-Driven Architecture*). W przeciwieństwie do klasycznego modelu żądanie-odpowiedź (*Request-Response*), typowego dla statycznych stron WWW, model ten zakłada, że przepływ sterowania w systemie jest determinowany przez wystąpienie określonych zdarzeń (akcji użytkownika, zmian stanu czujników), a nie przez sekwenncyjny kod proceduralny.

**2.2.3.1 Zastosowanie w projekcie** W zrealizowanym systemie bramy IoT, architektura sterowana zdarzeniami została wykorzystana jako główny mechanizm komunikacji dwukierunkowej (*Full-Duplex*) między Warstwą Prezentacji a Warstwą Logiki. Zastosowanie tego wzorca było niezbędne do osiągnięcia nieskiego opóźnienia wymaganego przy zdalnym sterowaniu mechanicznym oraz do natychmiastowego powiadamiania użytkownika o zagrożeniach. Wzorzec ten obsługuje trzy kluczowe obszary funkcjonalne:

- **Sterowanie PTZ (Uplink):** Zdarzenia płynące od użytkownika do serwera, sterujące silnikami kamery.
- **Powiadomienia o Alarmach (Downlink):** Zdarzenia płynące z serwera do użytkownika, informujące o wykryciu ruchu przez algorytm analizy obrazu.

- **Zarządzanie Stanem Nagrywania:** Synchronizacja interfejsu użytkownika ze stanem procesu rejestracji wideo na serwerze.

Dzięki luźnemu powiązaniu komponentów (*loose coupling*), serwer może obsłużyć setki takich zdarzeń na sekundę, zapewniając płynne sterowanie.

**2.2.3.2 Zasada działania** Istotą EDA jest odwrócenie zależności komunikacyjnych. Komponenty systemu nie odpytują się wzajemnie o zmianę stanu (co generowałoby zbędny ruch sieciowy i opóźnienia), lecz oczekują naadejście sygnału. Wzorzec ten składa się z trzech głównych elementów:

**Producent Zdarzenia:** Komponent, który wykrywa zmianę (np. naciśnięcie przycisku, wykrycie ruchu) i emituje komunikat. Producent nie musi wiedzieć, kto i w jaki sposób obsługuje to zdarzenie.

**Kanał Zdarzeń:** Medium transportowe, które przekazuje zdarzenie od producenta do konsumenta.

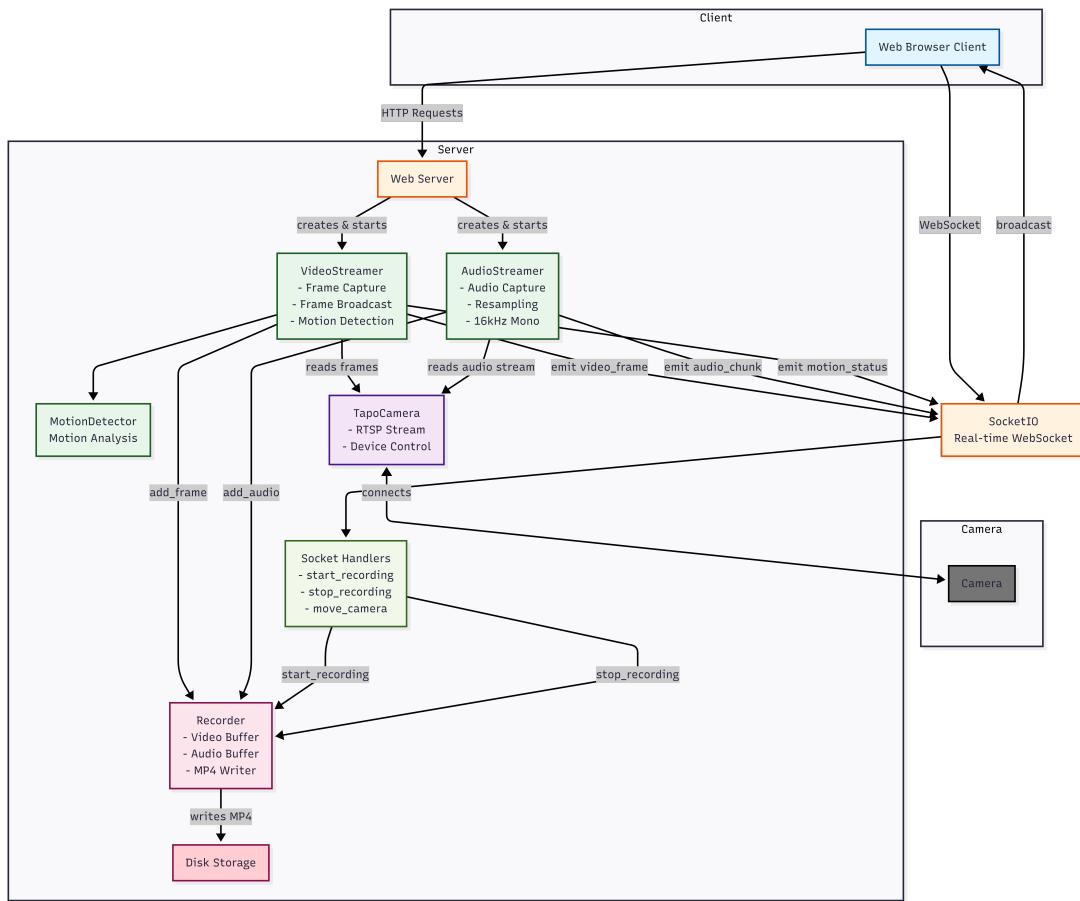
**Konsument Zdarzenia:** Komponent, który nasłuchuje na określony typ zdarzenia i w reakcji na nie uruchamia odpowiednią logikę biznesową.

## 2.3 Diagramy

Niniejsza sekcja zawiera kluczowe diagramy architektoniczne przedstawiające strukturę systemu i przepływ danych. Diagramy zostały opracowane z wykorzystaniem notacji UML oraz Mermaid.

### 2.3.1 Diagram architektury systemu

Poniższy diagram przedstawia ogólną architekturę systemu, ukazując przepływ danych między warstwą prezentacji, logiką aplikacji oraz dostępem do urządzenia.



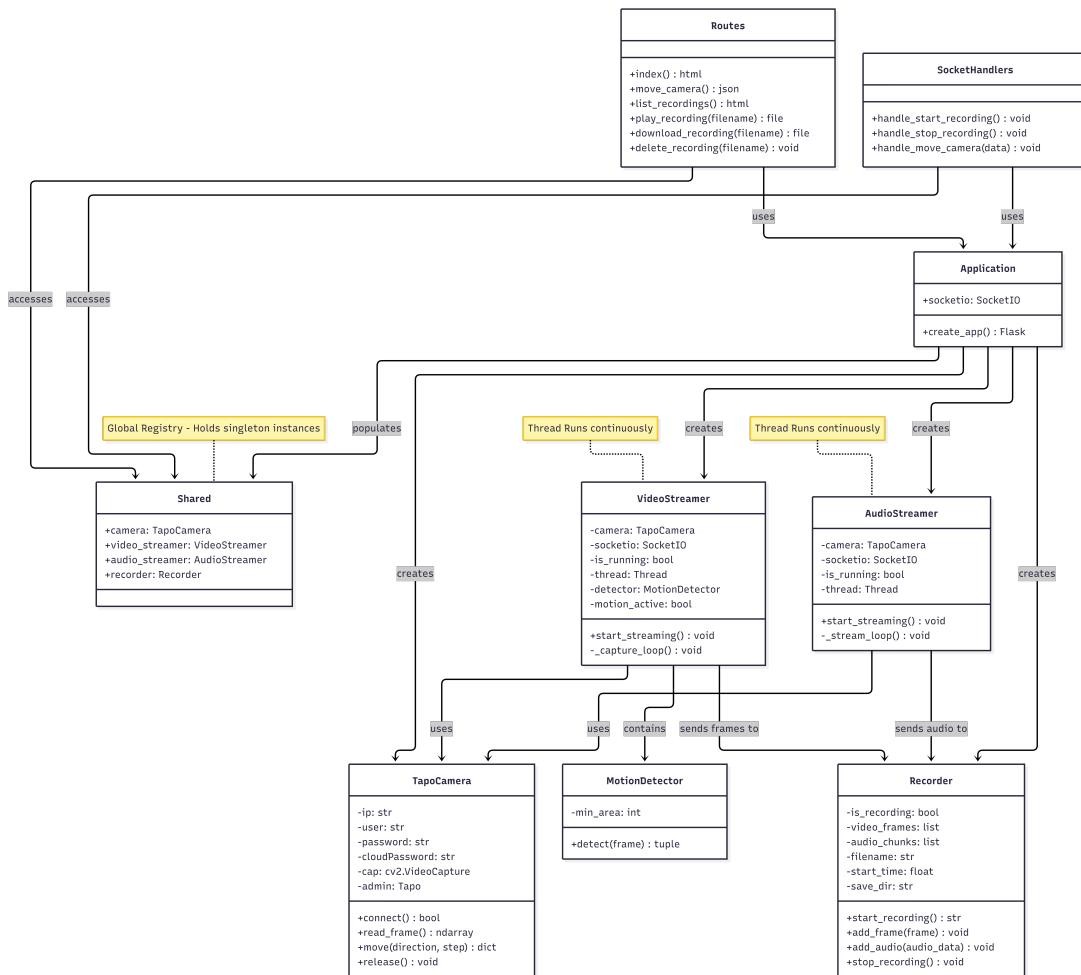
Rysunek 2.2: Architektura systemu IoT Gateway - przepływ danych między komponentami

### 2.3.2 Diagram klas

Diagram klas zobrazuje strukturę obiektową systemu, relacje między klasami oraz ich odpowiedzialności. Szczególnie istotne jest zrozumienie roli modułu `shared.py`, który pełni funkcję centralnego rejestru instancji.

#### Główne klasy i ich role:

- **TapoCamera:** Klasa odpowiadająca za komunikację z urządzeniem. Zarządza połączeniem RTSP dla streamu wideo oraz API Tapo dla sterowania PTZ.
- **VideoStreamer («Thread»):** Wątek odpowiadający za ciągłe odczytywanie klatek z kamery, przeprowadzanie analizy detektora ruchu, wysyłanie klatek do nagrywania oraz broadcast do klientów webowych. Współdzieli dostęp do instancji TapoCamera z wątkiem audio.
- **AudioStreamer («Thread»):** Wątek obsługujący strumieniowanie audio. Pobiera dane audio z kamery, przeprowadza resampling do 16 kHz w formacie mono, a następnie przesyła do nagrywania i broadcast.
- **Recorder:** Klasa bufferująca klatki wideo i fragmenty audio w pamięci RAM. Po



Rysunek 2.3: Diagram UML klas - struktura obiektowa aplikacji

zatrzymaniu nagrywania łączy dane multimedialne w plik MP4 z prawidłową synchronizacją audio-wideo.

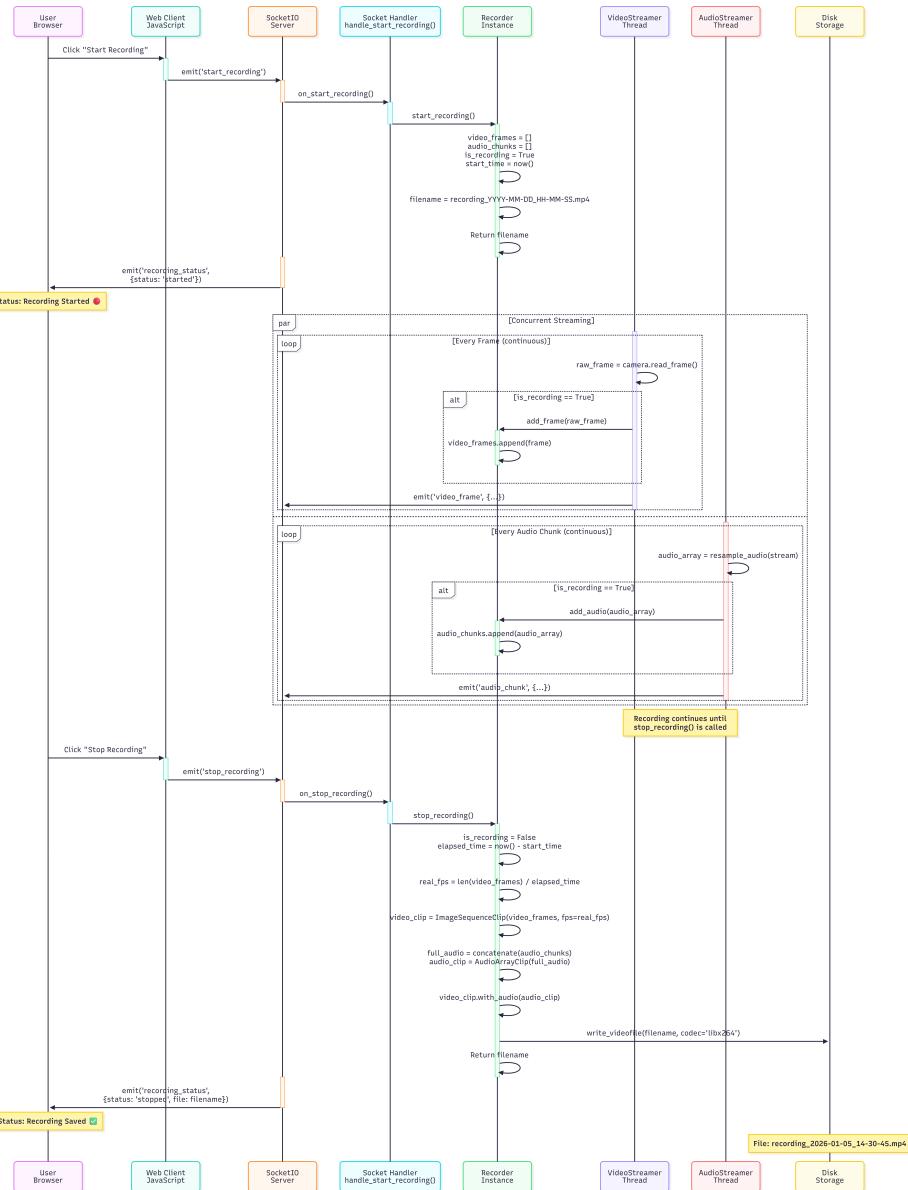
- **MotionDetector**: Klasa implementująca algorytm detekcji ruchu. Stosowana przez **VideoStreamer** do analizy każdej klatki w celu wygenerowania alertów.
- **Routes i SocketHandlers**: Komponenty warstwy webowej. **Routes** definiują punkty końcowe (*Endpoints*), a **SocketHandlers** obsługują zdarzenia WebSocket z klienta.
- **Shared (shared.py)**: Pseudo-klasa reprezentująca moduł globalny przechowujący instancje czterech głównych komponentów: **video\_streamer**, **audio\_streamer** oraz **recorder** i **camera**. Umożliwia dostęp do tych instancji z dowolnego miejsca w aplikacji bez konieczności przekazywania referencji.

#### Relacje między klasami:

- **Kompozycja**: **VideoStreamer** i **AudioStreamer** zawierają referencję do **TapoCamera**. Jest to związek agregacji, gdzie streamer „posiada” kamerę.
- **Zależność**: **VideoStreamer** i **AudioStreamer** są zależne od klasy **Recorder**, do której przesyłają dane za pośrednictwem metod **add\_frame()** i **add\_audio()**.

- **Dostęp do Shared:** Klasy Routes i SocketHandlers importują moduł shared.py w celu uzyskania dostępu do komponentów głównych bez bezpośredniego ich tworzenia.

### 2.3.3 Diagram sekwencji - Proces nagrywania



Rysunek 2.4: Diagram sekwencji - przebiegi procesu nagrywania

Powyższy diagram sekwencji ilustruje ścieżkę wykonania aplikacji od momentu kliknięcia przycisku "Start Recording" przez użytkownika aż do zapisania pliku na dysku. Diagram ukazuje zarówno sekwencyjne kroki jak i równoległe operacje (wątki VideoStreamer i AudioStreamer działające jednocześnie). **Najistotniejsze etapy:**

1. **Inicjalizacja:** Użytkownik kliką przycisk "Start Recording", wysyłając zdarzenie WebSocket do serwera.

2. **Aktywacja bufora:** Serwer wywoła metodę `start_recording()`, inicjalizując puste listy dla klatek wideo i fragmentów audio oraz ustawiając flagę `is_recording` = True.
3. **Strumienianie równoczesne:** Oba wątki (VideoStreamer i AudioStreamer) kontynuują swoją normalną pracę, ale teraz dodatkowo sprawdzają stan flagi `is_recording`. Jeśli jest True, przesyłają dane do bufora Recorder.
4. **Zatrzymanie:** Użytkownik kliką "Stop Recording", serwer ustawia `is_recording` = False, a Recorder rozpoczyna przetwarzanie zgromadzonych danych.
5. **Miksowanie i zapis:** Recorder łączy wszystkie klatki w sekwencję, łączy fragmenty audio, synchronizuje je z wideo i zapisuje plik MP4 na dysku.
6. **Potwierdzenie:** Serwer wysyła potwierdzenie do klienta, wyświetlając komunikat o pomyślnym zapisaniu pliku nagrania.

Diagram wyraźnie pokazuje krytyczną rolę współdzielonego stanu (`is_recording`) oraz wielowątkowości w realizacji funkcji nagrywania bez zakłócania podglądu na żywo.

## 2.4 Zastosowane narzędzia i technologie

Niniejszy rozdział stanowi wprowadzenie i analizę techniczną **stosu technologicznego** dobranego do realizacji projektu inżynierskiego, którego celem jest stworzenie otwartego systemu obsługi kamer IoT, na przykładzie modelu TP-Link Tapo C200.

W poniższych podrozdziałach dokonano dekonstrukcji architektury systemu na poziomie narzędziowym, omawiając zarówno warstwę językową, środowiskową, jak i bibliotek przetwarzania sygnałów.

### 2.4.1 Język programowania

Wybór języka **Python** w wersji **3.13** jako fundamentu warstwy logicznej projektu stanowił decyzję strategiczną, wynikającą z analizy wymagań stawianych współczesnym systemom IoT oraz aplikacjom przetwarzającym multimedia. W kontekście inżynierii oprogramowania, dobór technologii musi uwzględniać wypadkową dostępnych narzędzi oraz skalowalności rozwiązania.

### Bogactwo ekosystemu bibliotecznego i interoperacyjność

Kluczowym argumentem przemawiającym za wyborem tego środowiska jest dostępność i stabilność zaawansowanych bibliotek dedykowanych przetwarzaniu sygnałów. W ekosystemie Pythona możliwe jest wykorzystanie gotowych, wysoce zoptymalizowanych wrapperów na biblioteki natywne. Moduły takie jak `threading` pozwalają na

efektywne zarządzanie operacjami wejścia/wyjścia, co jest krytyczne dla zachowania płynności strumieniowania w czasie rzeczywistym.

## Szybkie prototypowanie i paradygmat Rapid Application Development (RAD)

Specyfika pracy inżynierskiej wymaga narzędzi umożliwiających szybką iterację i weryfikację hipotez. Python, jako język dynamicznie typowany o wysokiej ekspresywności składni, drastycznie skraca cykl twórczy oprogramowania. W kontekście integracji z urządzeniami IoT, pozwala to na elastyczne dostosowywanie protokołów komunikacyjnych i logiki sterowania bez konieczności długotrwałej rekompilacji całego projektu.

### 2.4.2 Zarządzanie zależnościami

W inżynierii oprogramowania systemów wbudowanych, **stabilność i powtarzalność środowiska** są kluczowe. Tradycyjne narzędzia zarządzania pakietami w Pythonie, takie jak pip, często zawodzą w złożonych scenariuszach CI/CD ze względu na wolny proces rozwiązywania zależności (*dependency resolution*) i brak determinizmu.

#### Nowoczesne Narzędzie: uv

W projekcie zastosowano uv – nowoczesny menedżer pakietów napisany w języku **Rust**. Narzędzie zostało wybrane ze względu na swoją bezkompromisową **wydajność**. Benchmarki wskazują, że uv potrafi instalować pakiety i rozwiązywać drzewa zależności od 10 do 100 razy szybciej niż standardowy pip. Skrócenie tego czasu znacząco przyspiesza cykl deweloperski w kontekście budowania obrazów **Docker**.

#### Determinizm i Pliki Blokady

Kluczowym aspektem dla pracy inżynierskiej jest gwarancja, że system wdrożony na urządzeniu produkcyjnym będzie posiadał identyczne wersje bibliotek co środowisko deweloperskie. uv wprowadza obsługę uniwersalnych **plików blokady** (uv.lock), które precyzyjnie definiują całe drzewo zależności wraz z **sumami kontrolnymi** (hashes), gwarantując kryptograficzną spójność środowiska. Jest to mechanizm podnoszący standard inżynierijny projektu, analogiczny do Cargo.lock w Rust.

#### Kompatybilność z Dockerem

Narzędzie uv zostało zaprojektowane z myślą o integracji z **konteneryzacją**. Posiada natywną obsługę generowania zoptymalizowanych plików zależności na podstawie plików blokady, co umożliwia szybkie i powtarzalne budowanie obrazów Docker bez konieczności ponownego rozwiązywania zależności za każdym razem.

### 2.4.3 Ekosystem konteneryzacji

Wdrożenie oprogramowania na **urządzeniach brzegowych** wiąże się z wyzwaniami heterogeniczności sprzętowej. Zastosowanie technologii **Docker** w niniejszym projekcie nie jest jedynie wygodą, lecz koniecznością architektoniczną zapewniającą **izolację, przenośność i bezpieczeństwo**.

#### Izolacja Procesów i Bezpieczeństwo

Kamery IoT, w tym modele **Tapo**, operują w strefie podwyższzonego ryzyka cybernetycznego. Uruchomienie autorskiego serwera sterującego bezpośrednio na systemie operacyjnym hosta niosłoby ryzyko, że ewentualne przejęcie kontroli nad aplikacją dąłoby atakującemu dostęp do całego systemu. Docker zapewnia silną **izolację procesów**.

#### Multi-stage Builds i Optymalizacja Rozmiaru

Urządzenia klasy *embedded* często dysponują ograniczoną przestrzenią dyskową. Aby pogodzić wymagania posiadania ciężkich narzędzi komplikacji (np. GCC, numpy) z koniecznością lekkiego obrazu końcowego, zastosowano technikę **budowania wieloetapowego (Multi-stage Builds)**.

1. **Stage 1 (Builder):** Obraz zawierający pełny *toolchain* (kompilatory GCC, nagłówki systemowe, uv, git).
2. **Stage 2 (Runtime):** Obraz typu „slim” (np. `python:3.13-slim-bookworm`), do którego kopowane są jedynie wynikowe artefakty z etapu pierwszego.

Dzięki temu podejściu, finalny obraz kontenera jest pozbawiony zbędnych plików tymczasowych, *cache'u* i narzędzi deweloperskich, osiągając rozmiar rzędu **200-300 MB** zamiast ponad 1 GB, co przyspiesza jego dystrybucję i aktualizację.

### 2.4.4 Interfejs webowy i protokół komunikacji

Efektywna interakcja użytkownika z systemem IoT wymaga **warstwy prezentacji**, która jest w stanie obsłużyć dynamiczny charakter **danych strumieniowych**. W tradycyjnym modelu webowym, opartym na **bezstanowym protokole HTTP (Request-Response)**, realizacja płynnego sterowania w czasie rzeczywistym jest nieefektywna. W związku z tym, w projekcie zastosowano architekturę opartą na **serwerze aplikacyjnym Flask** oraz **protokole WebSocket**, umożliwiającym dwukierunkową komunikację w czasie rzeczywistym.

## Serwer Aplikacyjny: Flask

Wybrano **Flask – lekki mikro-framework** w Pythonie (zgodny ze standardem *WSGI*). W przeciwieństwie do rozwiązań typu „full-stack”, Flask nie narzuca sztywnej struktury. Posiada minimalny narzut pamięciowy i pełni rolę lekkiego klienta serwerowego, odpowiedzialnego za:

- **Orkiestrację wątków:** Integracja asynchronicznych bibliotek sterujących kamerą.
- **Routing:** Obsługa statycznych plików interfejsu oraz końcówek API (*endpoints*).

## Protokół Transportowy: WebSockets

Zastosowano protokół **WebSocket** (*RFC 6455* Fette i Melnikov, 2011) przy użyciu biblioteki `Flask-SocketIO`. Zapewnia on zestawienie **trwałego, dwukierunkowego kanału komunikacji** (pełny dupleks) między przeglądarką klienta a serwerem, eliminując opóźnienia wynikające z cyklicznego odpytywania (*polling*).

Umożliwia to realizację dwóch celów:

- **Transmisja Wideo:** Klatki wideo są przesyłane jako binarne ładunki przez otwarty socket, co pozwala na redukcję opóźnień transmisji.
- **Sterowanie Czasu Rzeczywistego:** Komendy sterujące **PTZ** są przesyłane jako lekkie obiekty **JSON**.

## Warstwa Klienta (Frontend): Vanilla HTML/JS

W warstwie interfejsu użytkownika podjęto świadomą decyzję o rezygnacji z rozbudowanych frameworków JavaScript (np. React, Vue) na rzecz **natywnych technologii webowych: Vanilla JavaScript, HTML5** oraz CSS3. Zastosowanie **czystego JavaScriptu** pozwoliło na:

- **Maksymalną wydajność renderowania:** Bezpośrednia manipulacja drzewem DOM jest szybsza niż mechanizmy wirtualnego DOM występujące w frameworkach.
- **Redukcję długości technologicznej:** Kod klienta nie wymaga procesu kompilacji.

### 2.4.5 Biblioteki przetwarzania multimedialnych

W projektowanym systemie nadzoru wizyjnego, kluczową rolę technologiczną odgrywa biblioteka **OpenCV** (*Open Source Computer Vision Library*). Stanowi ona rdzeń analityczny, odpowiadając za **akwizycję strumienia wideo** z kamer TP-Link Tapo oraz

jego zaawansowaną **analizę w czasie rzeczywistym**. Biblioteka **PyAV** została wprowadzona jako rozwiązanie komplementarne, dedykowane wyłącznie do obsługi **ścieżki dźwiękowej**.

## OpenCV jako główny silnik wideo i analityczny

Decyzja o uczynieniu OpenCV główną biblioteką projektu podkutowana była jej pozycją jako **standardu przemysłowego** oraz kompleksowością oferowanych rozwiązań. W ramach opracowanego oprogramowania, OpenCV realizuje pełen cykl życia danych wizyjnych:

- **Akwizycja Obrazu:** Wykorzystanie interfejsu `cv2.VideoCapture` pozwala na **stabilne nawiązanie połączenia** ze strumieniem **RTSP** kamery.
- **Przetwarzanie Macierzowe i Analityka:** Po pobraniu klatki, OpenCV jest w stanie wykonywać na niej szereg operacji:
  - **Konwersje Kolorów:** Przekształcanie przestrzeni barw (np. BGR do HSV) w celu ułatwienia analizy.
  - **Filtracja i Wygładzanie:** Zastosowanie filtrów Gaussa w celu redukcji szumów.
  - **Detekcja Ruchu:** Implementacja algorytmów opartych na odejmowaniu tła i progowaniu różnic.
- **Optymalizacja:** Dzięki **backendowi napisanemu w C++**, OpenCV zapewnia wysoką wydajność operacji na macierzach, co jest kluczowe przy przetwarzaniu obrazu o wysokiej rozdzielczości na urządzeniach o ograniczonej mocy obliczeniowej.

## PyAV: Uzupełnienie luki funkcjonalnej (Audio)

Mimo wszechstronności w dziedzinie wideo, **OpenCV** posiada ograniczenia w zakresie **obsługi dźwięku** – biblioteka ta całkowicie ignoruje pakiety audio przesyłane w kontenerze RTSP.

W celu rozwiązania tego problemu inżynierskiego zastosowano bibliotekę **PyAV**. Jej rola w projekcie jest ściśle zdefiniowana i ograniczona do: **Równoległego nawiązania połączenia, Ekstrakcji, dekodowania i transkodowania strumienia audio** (z formataw PCM/AAC), przy jednoczesnym ignorowaniu pakietów wideo w celu oszczędności zasobów CPU.

Taka architektura pozwala na wykorzystanie pełnej mocy OpenCV do analizy obrazu, delegując jedynie niezbędne minimum (obsługę mikrofonu) do wyspecjalizowanej biblioteki PyAV.

Tabela 2.2: Podział kompetencji w warstwie multimedialnej

Biblioteka	Status w projekcie	Odpowiedzialność
OpenCV	Główna ( <i>Core</i> )	Pobieranie wideo (RTSP), dekodowanie obrazu, detekcja ruchu, nanoszenie OSD, przygotowanie klatek do streamingu.
PyAV	Pomocnicza ( <i>Auxiliary</i> )	Przechwytywanie wyłącznie ścieżki audio, transkodowanie dźwięku.

#### 2.4.6 Kontrola kamery

Realizacja nadzawanego celu pracy – **pełnego uniezależnienia systemu monitoringu od infrastruktury chmurowej producenta** – wymagała rozwiązania problemu **zamkniętej architektury** urządzenia. Kamera TP-Link Tapo C200 nie udostępnia publicznego **API** dla sieci lokalnej (*LAN*), co jest klasycznym przykładem strategii ***Vendor Lock-in***.

Aby przełamać to ograniczenie, w warstwie sterowania wykorzystano bibliotekę **PyTapo**. Jest to rozwiązanie typu **Open Source**, stanowiące implementację klienta własnościowego protokołu komunikacyjnego TP-Link, powstałe w wyniku procesów **inżynierii wstecznej** (*Reverse Engineering*).

#### Mechanizm działania i emulacja klienta

Działanie biblioteki opiera się na **symulacji zachowania oficjalnej aplikacji mobilnej**. Analiza ruchu sieciowego wykazała, że kamera wykorzystuje zmodyfikowany protokół **HTTP** do przesyłania danych sterujących w formacie **JSON**. Komunikacja ta jest załączona na kilku poziomach, które **PyTapo** skutecznie emuluje:

- **Negocjacja sesji (*Handshake*):** Biblioteka implementuje złożony proces **uwierzytelniania**, wymagający wymiany **kluczy sesyjnych** oraz **tokenów** (*stok*), generowanych w oparciu o algorytmy skrótu (MD5/SHA) i nonce.
- **Szyfrowanie Payloadu:** W przeciwieństwie do otwartych standardów, parametry sterujące (np. koordynaty silnika PTZ) nie są przesyłane jawnym tekstem. PyTapo implementuje algorytmy **szyfrowania symetrycznego** (warianty AES), co pozwala na konstruowanie poprawnych, zaszyfrowanych zapytań.

## Przewaga nad standardem ONVIF

Wybór PyTapo był podyktowany ograniczeniami implementacyjnymi standardu **ONVIF** (*Open Network Video Interface Forum*), który w tanich kamerach Tapo ogranicza się często tylko do strumieniowania wideo (*RTSP*).

Zastosowanie PyTapo umożliwiło dostęp do „ukrytych” **funkcji administracyjnych**, niedostępnych przez generyczne sterowniki:

- **Pełna kontrola PTZ** (*Pan-Tilt-Zoom*): Precyzyjne sterowanie silnikami krokowymi kamery.
- **Zarządzanie sensorem**: Programowe przełączanie trybu nocnego (**kontrola filtra IR-Cut**) oraz regulacja czułości detekcji ruchu.
- **Funkcje prywatności**: Możliwość zdalnego **wygaszenia obiektywu** (*Privacy Mode*) lub wyłączenia diody statusu LED.
- **Formatowanie nośników**: Zdalne zarządzanie kartą SD.

### 2.4.7 Narzędzie do kompozycji i zapisu danych

Ostatnim ogniwem w łańcuchu przetwarzania danych multimedialnych jest moduł odpowiedzialny za **trwały zapis** (*persistencję*) materiału dowodowego. Ze względu na przyjętą **architekturę hybrydową**, w której obraz i dźwięk przetwarzane są przez niezależne biblioteki (**OpenCV** i **PyAV**), zaistniała konieczność zastosowania narzędzia efektywnie integrującego te dwa rozłączne strumienie. Do realizacji tego zadania wybrano bibliotekę **MoviePy**.

## Rola integratora strumieni (Multipleksing)

MoviePy pełni w projekcie funkcję **orkiestratora procesu zapisu**. Jego zadaniem jest przeprowadzenie **multipleksowania** danych wizyjnych z OpenCV i próbek audio z PyAV zgromadzonych w buforach pamięci. Wynikiem jest **enkapuliacja** do standartowego **kontenera multimedialnego** (MP4) z kodekami **H.264** i **AAC**. Wybór dedykowanej biblioteki gwarantuje zachowanie **spójności struktury pliku wynikowego**.

## Abstrakcja nad FFmpeg i Synchronizacja A/V

MoviePy to wysokopoziomowa nakładka na oprogramowanie **FFmpeg**. FFmpeg to potężne narzędzie do przetwarzania multimediiów, które obsługuje szeroki zakres formatów i operacji na strumieniach audio i wideo. Zastosowanie jej eliminuje złożoność bezpośredniego wywoływania komend FFmpeg i zapewnia automatyczną **synchronizację A/V**, zarządzając osią czasu i dopasowując długość ścieżki audio do sekwencji

wideo. To kluczowe w przypadku **detekcji ruchu**, gdzie nagrania mają zmienną długość.

## 2.5 Proces implementacji rozwiązania

Niniejszy podrozdział stanowi techniczną dokumentację procesu transformacji koncepcji architektonicznej, zdefiniowanej w sekcji 3.4, w pełni funkcjonalny **artefakt programistyczny**. Celem poniższego opisu jest przedstawienie iteracji ewolucyjnej **cyklu wytwarzania oprogramowania**, który doprowadził do powstania prototypu systemu integrującego zamknięty ekosystem kamer TP-Link Tapo z otwartym środowiskiem *Open Source*.

Proces implementacji został podzielony na etapy odzwierciedlające **warstwową strukturę** projektowanego systemu, począwszy od najniższej **warstwy sprzętowej** (*Hardware Abstraction Layer*), poprzez **logikę biznesową** (*Middleware*), aż po **warstwę prezentacji** (*Frontend*). Takie podejście pozwoliło na empiryczną weryfikację założeń o trójwarstwowej i potokowej architekturze rozwiązania, zapewniając jednocześnie izolację poszczególnych modułów i łatwość ich późniejszego testowania.

W kolejnych punktach przedstawiono szczegółowo sposób rozwiązania kluczowych **wyzwań inżynierskich**, takich jak:

- nawiązanie stabilnego połączenia z urządzeniem **IoT** operującym na **własnościowych protokołach**,
- implementacja **asynchronicznego przetwarzania** strumieni multimedialnych (audio/wideo) w czasie rzeczywistym przy użyciu bibliotek **OpenCV** i **PyAV**,
- opracowanie **algorytmów detekcji ruchu**,
- **konteneryzacja aplikacji** z wykorzystaniem środowiska **Docker**, zapewniająca jej przenośność i powtarzalność wdrożenia.

Opisany poniżej proces stanowi syntezę doboru odpowiednich narzędzi oraz metodologii inżynierskiej, prowadzącą do uzyskania gotowego narzędzia nadzoru wizjnego, niezależnego od chmury producenta.

### 2.5.1 Provisioning i pierwotna konfiguracja środowiska kamery

Proces implementacji rozwiązania rozpoczęto od fizycznego uruchomienia urządzenia oraz przeprowadzenia procedury **provisioningu** (*wstępnej konfiguracji*), mającej na celu włączenie kamery do lokalnej infrastruktury sieciowej i odblokowanie interfejsów komunikacyjnych niezbędnych do tworzonego oprogramowania.

## **Stan faktyczny i ograniczenia fabryczne**

Analiza fabrycznie nowego urządzenia TP-Link Tapo C200 wykazała, że funkcjonuje ono w modelu tzw. „zamkniętego ogrodu” (*walled garden*). Domyślana konfiguracja *firmware'u* jest nastawiona wyłącznie na komunikację z chmurą producenta. W stanie „po wyłączeniu z pudełka” kamera charakteryzuje się następującymi ograniczeniami:

- Zablokowanie dostępu do strumieniowania protokołem **RTSP**.
- Brak możliwości sterowania kamerą.
- Brak połączenia z siecią lokalną.
- Brak zdefiniowanego lokalnego użytkownika administracyjnego, co uniemożliwia autoryzację zewnętrznych skryptów sterujących.

## **Procedura konfiguracji i odblokowania dostępu**

W celu przystosowania urządzenia do współpracy z autorskim rozwiązańiem *Open Source*, przeprowadzono następującą **sekwencję działań konfiguracyjnych**:

1. **Inicjalizacja sieciowa:** Wykorzystując oficjalną aplikację mobilną Tapo, nawiązano tymczasowe połączenie i przekazano poświadczenia docelowej sieci Wi-Fi.
2. **Utworzenie konta lokalnego:** W ustawieniach zaawansowanych aplikacji mobilnej zdefiniowano dedykowane konto lokalne. Jest to krok **krytyczny**, ponieważ utworzone w ten sposób login i hasło (`TAPO_USERNAME` i `TAPO_PASSWORD`) są przechowywane w pamięci urządzenia i służą do późniejszej autoryzacji zapytań **RTSP** oraz **HTTP** wysyłanych przez aplikację.
3. **Aktywacja interfejsów otwartych:** Wymuszono tryb zgodności z oprogramowaniem zewnętrznym poprzez włączenie opcji obsługi „Third-party software”. Operacja ta skutkowała otwarciem portu **554** (dla strumienia wideo RTSP).

Przeprowadzenie powyższych kroków pozwoliło na transformację urządzenia z pasywnego klienta chmury w **aktywny węzeł sieciowy**, gotowy do przyjmowania poleceń i udostępniania mediów poprzez standardowe protokoły sieciowe.

### **2.5.2 Konfiguracja środowiska programistycznego**

Przygotowanie środowiska deweloperskiego rozpoczęto od inicjalizacji projektu przy użyciu narzędzia `uv`. Proces ten przebiegał w dwóch głównych fazach: utworzenia podstawowej struktury (*scaffolding*) oraz instalacji i zablokowania zależności bibliotecznych.

## Iinicjalizacja i zarządzanie zależnościami

W pierwszej kolejności, w pustym katalogu roboczym, wykonano polecenie `uv init`. Operacja ta wygenerowała podstawowe pliki konfiguracyjne, w tym plik `.python-version`, w którym sztywno zdefiniowano wersję interpretera na **Python 3.13**.

Następnie przystąpiono do instalacji wymaganych bibliotek zewnętrznych za pomocą polecenia `uv add`:

- `flask` oraz `flask-socketio` – obsługa serwera HTTP i komunikacji WebSocket.
- `opencv-python-headless` – przetwarzanie obrazu (wersja zoptymalizowana dla serwerów).
- `av` – demultipleksowanie i dekodowanie audio/video (nakładka FFmpeg) dla obsługi ścieżki audio.
- `pytapo` – biblioteka kliencka do komunikacji z kamerą.
- `moviepy` oraz `numpy` – operacje na plikach video i macierzach danych.

Efektem tego procesu było wygenerowanie plików `pyproject.toml` (deklaracja zależności) oraz `uv.lock` (drzewo zależności ze **skrótami kryptograficznymi**), co zamknęło etap konfiguracji środowiska uruchomieniowego.

## Struktura projektu

Na bazie zainicjowanego środowiska utworzono docelową strukturę katalogów i plików, która dzieli aplikację na logiczne moduły funkcjonalne. Architektura plików w projekcie prezentuje się następująco:

- **Katalog główny (Root):**

- `run.py`: Główny punkt wejścia (*entry point*) uruchamiający serwer.
- `Dockerfile` oraz `uv.lock/pyproject.toml`.

- **Moduł aplikacji (app/):**

- `init.py`: Fabryka aplikacji Flask, inicjującainstancję serwera.
- `settings.py` oraz `shared.py`: Moduły do ładowania konfiguracji i współdzielienia obiektów między wątkami.
- `camera/`: Warstwa abstrakcji sprzętowej (*HAL*) obsługująca bezpośrednie połączenie z urządzeniem.
- `video/, audio/, detection/, recording/`: Logika biznesowa i przetwarzanie mediów/analiza obrazu.

- `web/`: Warstwa prezentacji zawierająca trasy (*routes*), obsługę zdarzeń `WebSocket` oraz szablony HTML (*templates*).

Tak przygotowana struktura zapewniła separację logiki biznesowej od warstwy sprzętowej i interfejsu użytkownika, co było niezbędne do dalszej implementacji poszczególnych funkcjonalności.

### 2.5.3 Implementacja serwera HTTP

Centralnym punktem logicznym systemu, spajającym warstwę prezentacji z logiką backendową, jest serwer HTTP zrealizowany w oparciu o mikroframework **Flask**.

#### Rozwiążanie problemu „Double Execution”

Wyzwaniem na etapie implementacji serwera było dostosowanie jego cyklu życia do **ograniczeń sprzętowych** kamery TP-Link Tapo C200, która limituje liczbę jednocześnie działających sesji **RTSP** (zazwyczaj do dwóch).

Standardowy tryb deweloperski frameworka Flask wykorzystuje mechanizm *reloader*, który monitoruje zmiany w kodzie i powoduje uruchomienie **dowóch procesów systemowych**: procesu monitorującego oraz procesu roboczego. W kontekście aplikacji IoT skutkowało to podwójnym uruchomieniem wątków strumieniujących. Efektem było natychmiastowe wyczerpanie puli dostępnych połączeń kamery i odrzucanie prób autoryzacji.

W celu wyeliminowania tego błędu, w głównym punkcie wejścia aplikacji (`run.py`) wymuszono konfigurację serwera z ustawieniem parametru `use_reloader=False`. Zapewniło to deterministyczne, jednokrotne uruchomienie wątków `VideoStreamer` i `AudioStreamer`, gwarantując stabilność połączenia z kamerą przy zachowaniu pełnej kontroli nad zasobami sieciowymi.

### 2.5.4 Implementacja warstwy dostępu do sprzętu

**Warstwa Dostępu do Sprzętu** (ang. *Hardware Abstraction Layer – HAL*) stanowi fundamentalny element architektury systemu, izolujący wysokopoziomową logikę biznesową od specyfiki protokołów komunikacyjnych urządzenia końcowego. W projekcie funkcję tę pełni moduł `app/camera`, którego centralnym komponentem jest klasa `TapoCamera`.

Klasa ta realizuje **wzorzec fasady**, ukrywając złożoność obsługi strumienia **RTSP** oraz **API** urytego przez producenta oprogramowania dzięki bibliotece `PyTapo`. Dzięki takiemu podejściu, pozostałe moduły systemu operują na jednolitym interfejsie obiektowym, nie wymagając znajomości niskopoziomowych detali implementacyjnych.

## Inicjalizacja połączenia i akwizycja wideo

Nawiązanie komunikacji z kamerą odbywa się dwutorowo. Metoda `connect()` inicjuje niezależne sesje dla **podsystemu wideo** oraz **podsystemu sterowania**.

Adres strumienia budowany jest dynamicznie w oparciu o poświadczenia, zgodnie ze schematem `rtsp://user:password@ip/stream1`. Połączenie z serwerem RTSP realizowane jest za pomocą interfejsu `cv2.VideoCapture` z biblioteki OpenCV, który obsługuje protokół i dekodowanie strumienia wideo.

```
1  def connect(self):
2      # 1. Setup Video (RTSP)
3      url = f"rtsp://{{self.user}}:{{self.password}}@{{self.ip}}/
4          stream1"
5      self.cap = cv2.VideoCapture(url)
6      video_success = self.cap.isOpened()
7
8      # 2. Setup Controls (Pytapo)
9      try:
10         self.admin = Tapo(self.ip, self.user, self.
11             cloudPassword)
12         # Try a simple command to verify connection
13         self.admin.getBasicInfo()
14         control_success = True
15     except Exception as e:
16         print(f"Control Error: {e}")
17         control_success = False
18
19     if video_success and control_success:
20         print(f" Fully Connected to {{self.ip}}")
21         return True
22     else:
23         print(f" Connection Issues: Video={{video_success}}
24             , Controls={{control_success}}")
25         return False
```

Listing 2.1: Implementacja metody `connect()` w klasie `TapoCamera`

### 2.5.5 Abstrakcja sterowania mechaniką (PTZ)

Implementacja sterowania mechaniką PTZ została zrealizowana jako wrapper nad biblioteką PyTapo. Interfejs `move(direction, step)` mapuje semantyczne polecenia na

**wektor przesunięcia** silników ( $dx, dy$ ), przekazując **kierunek** i **krok** jako parametry i przeliczając je na wartości osi. Dzięki temu silnik wykonuje ruch o zadany krok w wskazanym kierunku, a logika pozostaje spójna z **fizycznymi ograniczeniami** urządzenia.

Kluczowym elementem jest **obsługa błędów kamery i mechanizm wyjątków** przy **przekroczeniu zakresu ruchu**. Warstwa PTZ przechwytuje wyjątki generowane przez PyTapo i rozpoznaje sytuacje przekroczenia limitów (np. słowa kluczowe „range”, „limit”), zwracając kontrolowany wynik operacji zamiast przerwać pracę systemu. W przypadku innych błędów podejmowana jest próba ponownego zestawienia połączenia z kamerą, a klient API otrzymuje **znormalizowany komunikat**.

Realizacja opiera się bezpośrednio na PyTapo: translacja kierunku na wektor przesunięcia polega na wywołaniu `moveMotor(dx, dy)`, gdzie wartości  $dx, dy$  wyznaczane są z pary `direction, step`: „up”  $\rightarrow (0, step)$ , „down”  $\rightarrow (0, -step)$ , „left”  $\rightarrow (-step, 0)$ , „right”  $\rightarrow (step, 0)$ .

## 2.5.6 Realizacja strumieniowania wideo

Za dystrybucję obrazu w czasie rzeczywistym odpowiada klasa `VideoStreamer` (moduł `app/video/streamer.py`). Jej implementacja opiera się na **modelu asynchronicznym**, wykorzystującym dedykowany **wątek systemowy** do cyklicznego pobierania i przetwarzania klatek obrazu, co zapewnia niezakłóconą pracę głównego serwera aplikacji.

### Architektura wątkowa i inicjalizacja

Zarówno w przypadku klasy `VideoStreamer` i `AudioStreamer` inicjowane są z referencjami do obiektu kamery (*warstwa HAL*) oraz instancji `socketio`. Uruchomienie procesu strumieniowania następuje poprzez metodę `start_streaming()`, która powołuje nowy wątek w trybie działania w tle. Taka konfiguracja gwarantuje, że proces strumieniowania zostanie automatycznie zakończony wraz z zamknięciem głównego procesu aplikacji.

```

1  class VideoStreamer:
2
3      def __init__(self, camera, socketio):
4          self.camera = camera
5          self.socketio = socketio
6          self.is_running = False
7          self.thread = None
8
9      # Inicjalizacja detektora ruchu
10     self.detector = MotionDetector(min_area=1000)
11     self.motion_active = False
12
13     def start_streaming(self):
14         if not self.is_running:
15             self.is_running = True
16             self.thread = threading.Thread(target=self.
17                 _capture_loop)
18             self.thread.daemon = True
19             self.thread.start()

```

Listing 2.2: Inicjalizacja i uruchomienie wątku VideoStreamer

## Potok przetwarzania obrazu

Rdzeń logiki strumieniowania zawarto w metodzie `_capture_loop()`. Realizuje ona sekwencyjny potok przetwarzania każdej klatki, składający się z pięciu kluczowych etapów: akwizycji, analizy, skalowania, kompresji oraz transmisji.

- Akwizycja i Dystrybucja Wewnętrzna:** Pętla pobiera surową klatkę z kamery za pomocą obiektu kamery z warstwy HAL. Jeżeli nagrywanie jest włączone, obraz jest natychmiast przekazywany do **Modułu Nagrywania** (`shared.recorder.add_frame`). Obraz jest analizowany przez **Modułu Detekcji** (`self.detector.detect`). Wynik analizy steruje emisją zdarzenia do klienta.
- Optymalizacja Transmisji:** W celu redukcji zużycia pasma sieciowego, obraz przeznaczony do podglądu jest skalowany w dół (np. do 1000x562 pikseli) przy użyciu `cv2.resize`, co jest kompromisem między jakością wizualną a opóźnieniem transmisji.
- Kompresja i Serializacja:** Przeskalowana klatka jest poddawana kompresji straatnej do formatu **JPEG** (`cv2.imencode`). Uzyskany bufor binarny jest następnie ko-

dowany do formatu **Base64** (`base64.b64encode`) dla bezpiecznego osadzenia w strukturze JSON przesyłanej przez WebSocket.

4. **Emisja i Taktowanie:** Gotowy ładunek danych jest wysyłany do klienta poprzez `socketio.emit`. Mechanizm taktowania (`socketio.sleep(0.04)`) ogranicza *frame rate* do około 25 klatek na sekundę, stabilizując obciążenie serwera.

```
1 def _capture_loop(self):
2     while self.is_running:
3         raw_frame = self.camera.read_frame()
4         if raw_frame is not None:
5             if shared.recorder and shared.recorder.
6                 is_recording:
7                 shared.recorder.add_frame(raw_frame)
8
9
10        is_motion, _ = self.detector.detect(raw_frame)
11
12        if is_motion != self.motion_active:
13            self.motion_active = is_motion
14            self.socketio.emit('motion_status', {'motion':
15                self.motion_active})
16
17        web_frame = cv2.resize(raw_frame, (1000, 562))
18        success, buffer = cv2.imencode('.jpg', web_frame)
19
20        if success:
21            b64_frame = base64.b64encode(buffer).decode(
22                'utf-8')
23            self.socketio.emit('video_frame', {'frame':
24                b64_frame})
25
26        self.socketio.sleep(0.04)
```

Listing 2.3: Potok przetwarzania video

### 2.5.7 Realizacja strumieniowania audio

O ile obsługa wideo opierała się na ustandaryzowanych mechanizmach biblioteki **OpenCV**, o tyle implementacja podsystemu audio wymagała rozwiązania szeregu problemów natury inżynierskiej, wynikających z braku natywnego wsparcia dla dźwięku w tej bibliotece. Podsystem audio zrealizowano w oparciu o klasę `AudioStreamer`, wykorzystując bibliotekę **PyAV** do bezpośredniej obsługi kontenera multimedialnego.

## **Problem niezgodności częstotliwości próbkowania**

Podczas wstępnych testów integracyjnych zidentyfikowano krytyczny błąd w reprodukcji dźwięku, określany jako zjawisko **przesunięcia widma** (potocznie „Demon Voice”). Wynikał on z niezgodności częstotliwości próbkowania między nadawcą kamerą, a odbiorcą - przeglądarką klienta.

- **Stan źródłowy:** Kamera Tapo przesyłała dźwięk w formacie wysokiej jakości 44.1 kHz.
- **Stan odbiorczy:** Prosty odtwarzacz PCM w przeglądarce klienta oczekiwany domyślnie strumienia o parametrach **16 kHz**.

Bezpośrednie przekazanie surowych danych powodowało, że przeglądarka „rozciągała” otrzymane próbki w czasie, co skutkowało około dwu- lub trzykrotnym zwolnieniem odtwarzania i drastycznym obniżeniem tonacji.

## **Implementacja Resamplingu i Normalizacji Dźwięku**

W celu wyeliminowania opisanych zniekształceń, w pętli przetwarzania audio zaimplementowano proces resamplingu. Wykorzystano klasę `av.AudioResampler`, wymuszając konwersję każdego pakietu do ścisłe zdefiniowanego formatu.

Kolejnym wyzwaniem była obsługa wielokanałowości, dlatego zastosowano **cyfrowe mikowanie kanałów**.

Algorytm oblicza **średnią arytmetyczną** z obu kanałów, tworząc zbalansowany sygnał monofoniczny. Dodatkowo, kluczowym krokiem była jawną konwersja typów danych z formatu *Float32* do wymaganego formatu *Int16*, co zapobiegało generowaniu silnego szumu statycznego po stronie klienta.

```

1      resampler = av.AudioResampler(format='s16', layout='mono',
2                                      , rate=16000)
3      for packet in container.demux(stream):
4          if not self.is_running:
5              break
6
7          for frame in packet.decode():
8              frame.pts = None
9              output_frames = resampler.resample(frame)
10
11         for out_frame in output_frames:
12             array = out_frame.to_ndarray()
13
14             # Mix to Mono if Stereo
15             if array.ndim == 2:
16                 if array.shape[0] > 1:
17                     array = np.mean(array, axis=0)
18                 elif array.shape[1] > 1:
19                     array = array.reshape(-1)
20
21             array = array.astype(np.int16)
22
23             self.socketio.emit('audio_chunk', array.tobytes())
24
25             if shared.recorder and shared.recorder.
is_recording:
26                 shared.recorder.add_audio(array)

```

Listing 2.4: Potok przetwarzania audio

Dzięki zastosowaniu powyższego potoku przetwarzania, uzyskano stabilny, zrozumiały sygnał audio o niskim opóźnieniu, kompatybilny z większością nowoczesnych przeglądarek internetowych.

### 2.5.8 Implementacja warstwy komunikacyjnej

Warstwa komunikacyjna (*Middleware*) w zaprojektowanym systemie pełni rolę dystrybutora danych, łączącego asynchroniczne procesy backendowe z interfejsem użytkownika. Ze względu na specyfikę aplikacji nadzoru wizyjnego, która wymaga jednocze-

snego przesyłania strumieni multimedialnych (*downlink*) oraz odbierania poleceń sterujących (*uplink*).

## Protokół WebSockets i Flask-SocketIO

Implementację oparto na bibliotece **Flask-SocketIO**, która zapewnia abstrakcję nad surowymi gniazdami sieciowymi. Pozwoliło to na zdefiniowanie dedykowanych kanałów komunikacyjnych dla różnych typów danych:

- `video_frame / audio_chunk`: Kanały wysokiej przepustowości do transmisji mediów.
- `motion_status`: Kanał zdarzeń asynchronicznych informujący o wykryciu ruchu.
- `control`: Kanał sterujący odbierający polecenia użytkownika.

## Model współbieżności (*Concurrency Model*)

Serwer aplikacji został zaprojektowany w modelu wielowątkowym, wykorzystując standardowy moduł `threading` języka Python. Architektura ta zakłada podział odpowiedzialności na:

- **Wątek Główny (*Main Thread*)**: Obsługuje pętlę zdarzeń serwera Flask, przyjmuje żądania HTTP oraz zarządza sesjami WebSocket.
- **Wątki Tła (*Daemon Threads*)**: Niezależne procesy lekkie, w których uruchomione są pętle VideoStreamer oraz AudioStreamer. Ich zadaniem jest ciągła akwizycja danych z kamery i ich emisja.

Taka separacja zapobiega blokowaniu interfejsu użytkownika w momentach intensywnego przetwarzania obrazu (detekcja ruchu) lub opóźnień w odpowiedzi kamery.

## Zarządzanie stanem współdzielonym

Wyzwaniem w środowisku wielowątkowym jest bezpieczny dostęp do zasobów. W projekcie zastosowano **wzorzec Singleton** realizowany poprzez moduł `app/shared.py`. Plik ten pełni funkcję **globalnej pamięci współdzielonej**, przechowując referencje do aktywnych instancji kamery, streamerów oraz rejestratora.

Dzięki temu rozwiązaniu, procedury obsługi zdarzeń (*Socket Handlers*) mogą wchodzić w interakcję z obiektami uruchomionymi w innych wątkach – na przykład, żądanie klienta o rozpoczęcie nagrywania może bezpośrednio wysterować obiekt Recorder, który jest zasilany danymi z wątku VideoStreamer.

Mechanizm ten skutecznie rozwiązuje problem komunikacji międzywątkowej w skali mikroserwisu obsługującego pojedyncze urządzenie IoT.

## 2.5.9 Budowa interfejsu użytkownika

**Warstwa prezentacji** (*Front-End*) została zrealizowana jako lekka aplikacja webowa. Jej głównym zadaniem jest wizualizacja strumieni multimedialnych z minimalnym opóźnieniem oraz zapewnienie responsywnego sterowania mechaniką kamery. Logikę klienta zaimplementowano w **JavaScript** z wykorzystaniem natywnych interfejsów przeglądarki (*HTML5 APIs*), bez udziału ciężkich frameworków frontendowych.

### Renderowanie Wideo: Canvas vs Video Tag

W klasycznych systemach monitoringu użycie znacznika HTML `<video>` narzuca wewnętrzne buforowanie przeglądarki, generując opóźnienia rzędu kilku sekund. W celu redukcji opóźnienia do rzędu milisekund, zastosowano alternatywne podejście oparte na elemencie `<canvas>` oraz protokole **WebSocket**.

Mechanizm ten działa następująco:

1. Klient otrzymuje zakodowaną w **Base64** ramkę obrazu poprzez zdarzenie `video_frame`.
2. Dane są ładowane do obiektu `Image()`.
3. Obraz jest natychmiastowo rysowany na elemencie `canvas` metodą `drawImage()`.

Pominięcie bufora odtwarzacza wideo pozwoliło na uzyskanie efektu czasu rzeczywistego, gdzie obraz widoczny na ekranie odpowiada aktualnemu stanowi sensora kamery.

### Reprodukcja Dźwięku i Jitter Buffer

Odtwarzanie dźwięku zrealizowano przy użyciu **Web Audio API**, co zapewniło niskopoziomową kontrolę nad potokiem audio. Surowe dane **PCM**, przesyłane z serwera, są konwertowane na `AudioBuffer` i kolejkowane do odtworzenia.

Wyzwaniem była kompensacja nierównomiernego dostarczania pakietów przez sieć (tzw. *Network Jitter*). Bezpośrednie odtwarzanie próbek skutkowało słyszalnymi trzaskami i przerwami. Rozwiązaniem było zaimplementowanie programowego **bufora** (*Jitter Buffer*). Algorytm ten dodaje stałe, minimalne opóźnienie (skonfigurowane na **0.05s**) do czasu startu każdego segmentu audio, co wygładza odtwarzanie bez zauważalnego wpływu na synchronizację z obrazem.

```

1 const JITTER_DELAY = 0.05;
2 let nextStartTime = 0;
3
4 socket.on('audio_chunk', function(data) {
5     // 1. Konwersja surowych bajtów na Float32
6     const int16Data = new Int16Array(data);
7     const float32Data = new Float32Array(int16Data.length);
8     for (let i = 0; i < int16Data.length; i++) {
9         // Normalizacja do zakresu -1.0 do 1.0
10        float32Data[i] = int16Data[i] / 32768;
11    }
12
13    // 2. Utworzenie bufora audio
14    const audioBuffer = audioCtx.createBuffer(1, float32Data.
15        length, 16000);
16    audioBuffer.getChannelData(0).set(float32Data);
17
18    // 3. Planowanie czasu odtworzenia (Scheduling)
19    const source = audioCtx.createBufferSource();
20    source.buffer = audioBuffer;
21    source.connect(audioCtx.destination);
22
23    // Algorytm Jitter Buffer:
24    const now = audioCtx.currentTime;
25    const playTime = Math.max(now + JITTER_DELAY,
26        nextStartTime);
27
28    source.start(playTime);
29    nextStartTime = playTime + audioBuffer.duration;
30 });

```

Listing 2.5: Implementacja kolejkowania audio (Jitter Buffer)

## 2.5.10 Detekcja zdarzeń

Kluczową funkcjonalnością systemu nadzoru, przekształcającą pasywny podgląd w aktywne narzędzie bezpieczeństwa, jest moduł analizy obrazu. W projekcie zaimplementowano algorytm **detekcji ruchu** działający na brzegu sieci (*Edge Processing*), bezpośrednio na serwerze aplikacji. Logikę tę zawarto w klasie MotionDetector.

## Algorytm adaptacyjnego modelowania tła

W przeciwnieństwie do prostych rozwiązań porównujących klatki sąsiednie (*Frame Differencing*), w projekcie zastosowano **model średniej ruchomej** (*Running Average*). Algorytm ten, realizowany przez funkcję `cv2.accumulateWeighted` (OpenCV), pozwala na **dynamiczną aktualizację modelu tła**.

Każda nowa klatka wpływa na wzorzec tła z określona wagą  $\alpha$ . Matematycznie proces ten opisuje równanie:

$$dst(x, y) = (1 - \alpha) \cdot dst(x, y) + \alpha \cdot src(x, y)$$

Gdzie  $src$  to klatka bieżąca, a  $dst$  to akumulowany model tła. Takie podejście sprawia, że system „przyzwyczaja się” do powolnych zmian oświetlenia, nie interpretując ich jako ruchu, co znacząco redukuje liczbę fałszywych alarmów.

## Potok przetwarzania i filtracja (*Pipeline*)

Proces detekcji przebiega w kilku sekwencyjnych etapach:

1. **Pre-processing:** Surowa klatka jest konwertowana do **skali szarości** (`cv2.cvtColor`), a następnie poddawana **rozmyciu Gaussa** (`cv2.GaussianBlur`). Operacja ta usuwa szum wysokoczęstotliwościowy.
2. **Wyznaczanie różnic (*Delta*):** Obliczana jest **bezwzględna różnica** (`cv2.absdiff`) pomiędzy bieżącą klatką a wyznaczonym modelem tła.
3. **Progowanie (*Thresholding*):** Obraz różnicowy jest **binaryzowany** (`cv2.threshold`). Piksele, których zmiana jasności przekroczyła ustalony próg (np. 25), oznaczone są jako ruch (wartość 255).
4. **Ekstrakcja konturów:** Na obrazie binarnym wyszukiwane są ciągłe obszary zmian za pomocą funkcji `cv2.findContours`.

```

1     def detect(self, frame):
2         small_frame = cv2.resize(frame, (500, 300))
3         gray = cv2.cvtColor(small_frame, cv2.COLOR_BGR2GRAY)
4         gray = cv2.GaussianBlur(gray, (21, 21), 0)
5
6         if self.avg_frame is None:
7             self.avg_frame = gray.copy().astype("float")
8             return False, {}
9
10        cv2.accumulateWeighted(gray, self.avg_frame, 0.5)
11
12        frame_delta = cv2.absdiff(gray, cv2.convertScaleAbs(
13            self.avg_frame))
14
15        thresh = cv2.threshold(frame_delta, 5, 255, cv2.
16            THRESH_BINARY)[1]
17        thresh = cv2.dilate(thresh, None, iterations=2)
18
19        contours, _ = cv2.findContours(thresh.copy(), cv2.
20            RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
21
22        motion_found = False
23
24        for c in contours:
25            if cv2.contourArea(c) < self.min_area:
26                continue
27            motion_found = True
28            break # We found at least one big movement
29
30        return motion_found, {}

```

Listing 2.6: Metoda detekcji ruchu wykorzystująca akumulację wagi

## Logika biznesowa i eliminacja zakłóceń

Ostatnim etapem jest weryfikacja wykrytych obiektów. System iteruje przez znalezione kontury, obliczając ich pole powierzchni za pomocą `cv2.contourArea`. Zdefiniowano **próg decyzyjny** `min_area` (domyślnie 5000 pikseli).

Kontury mniejsze od progu są ignorowane jako **szum** (np. poruszające się liście,

owady). Dopiero przekroczenie tej wartości skutkuje uznaniem zdarzenia za „Ruch”, co powoduje ustawienie flagi stanu na True.

Zastosowanie takiej kaskady filtrów (*Gaussian Blur* → *Accumulate Weighted* → *Area Threshold*) pozwoliło na uzyskanie stabilnego detektora, odpornego na typowe zakłócenia występujące w domowych systemach monitoringu.

### 2.5.11 Moduł rejestracji

Ostatnim ogniwem w łańcuchu przetwarzania danych jest moduł rejestracji, zaimplementowany w klasie `Recorder`. Jego zadaniem jest przechwycenie ulotnych strumieni wideo i audio oraz ich trwała archiwizacja w postaci **pliku multimedialnego**. Ze względu na wymagania dotyczące wydajności czasu rzeczywistego, zaprojektowano go w oparciu o strategię **odroczonego zapisu** (*Deferred Writing*).

#### Strategia buforowania w pamięci

W przypadku aplikacji działającej na sprzęcie o ograniczonej wydajności I/O, ciągłe operacje zapisu mogą prowadzić do **blokowania wątków i gubienia klatek**.

Aby wyeliminować to ryzyko, w projekcie zastosowano **buforowanie w pamięci operacyjnej RAM**. Podczas trwania nagrania, metody `add_frame()` oraz `add_audio()` nie wykonują operacji dyskowych, lecz jedynie dopisują przychodzące dane do list w pamięci (`self.video_frames`, `self.audio_chunks`).

Podejście to gwarantuje, że proces nagrywania nie wpływa negatywnie na **płynność podglądu na żywo** ani na działanie algorytmów detekcji ruchu.

#### Syntez pliku i Post-processing

Właściwy proces tworzenia pliku wideo jest inicjowany dopiero w momencie wywołania metody `stop_recording()`. Jest to operacja **post-processingu**, która wykorzystuje bibliotekę `MoviePy` do połączenia zebranych buforów w spójny strumień.

Proces ten składa się z trzech etapów:

1. **Konstrukcja klipu wideo:** Utworzenie obiektu `ImageSequenceClip` z listy zgromadzonych klatek RGB.
2. **Rekonstrukcja ścieżki dźwiękowej:** Scalenie fragmentów audio (np. `concatenate`) i utworzenie obiektu `AudioArrayClip`. Definiowana jest bazowa częstotliwość próbkowania strumienia wejściowego (**16000 Hz**).
3. **Renderowanie i Upsampling:** Zapis gotowego materiału na dysk za pomocą metody `write_videofile`.

Kluczowym zabiegiem inżynierskim jest **upsampling audio do 44.1 kHz** (`audio_fps=44100`). Eksperymenty wykazały, że wymuszenie standardu **CD-Quality** podczas renderingu eliminuje artefakty (np. „metaliczne brzmienie”) w niektórych odtwarzaczach systemowych, zapewniając szerszą kompatybilność nagrania.

### 2.5.12 Archiwizacja

Proces archiwizacji stanowi finalny etap potoku przetwarzania danych, w którym ulotne informacje zgromadzone w pamięci operacyjnej są przekształcane w trwałego pliku multi-medialnego. Za realizację tego zadania odpowiada metoda `stop_recording` klasy `Recorder`, która koordynuje syntezę strumieni wideo i audio.

### Przetwarzanie wstępne i buforowanie

W trakcie trwania nagrania system realizuje ciągłą akwizycję danych, wykonując niezbędne konwersje w czasie rzeczywistym:

- **Wideo (add\_frame):** Biblioteka OpenCV operuje w przestrzeni barw **BGR** (*Blue-Green-Red*), podczas gdy standardy kodowania wideo oczekują formatu **RGB**. Każda klatka przed dodaniem do bufora pamięci poddawana jest **permutacji kanałów**, co zapewnia poprawne odwzorowanie kolorów.
- **Audio (add\_audio):** Próbki dźwiękowe są agregowane w surowej postaci (*lista fragmentów PCM*), bez wstępnego przetwarzania, co minimalizuje narut obliczeniowy w trakcie nagrywania.

### Finalizacja i synchronizacja A/V

Kluczowe jest zapewnienie synchronizacji obrazu z dźwiękiem. W projekcie zastosowano metodę dynamicznego obliczania **rzeczywistego klatka**.

Zamiast zakładać stałą wartość FPS, system mierzy rzeczywisty czas trwania nagrania ( $T_{elapsed}$ ) oraz liczbę zgromadzonych klatek ( $N_{frames}$ ). Rzeczywista prędkość odtwarzania wyliczana jest ze wzoru:

$$FPS_{real} = \frac{N_{frames}}{T_{elapsed}}$$

### Synteza i zapis pliku

Proces zapisu realizowany jest z wykorzystaniem biblioteki **MoviePy** i przebiega w trzech fazach:

1. **Konkatenacja Audio:** Fragmenty dźwiękowe są łączone w jeden ciągły strumień.

2. **Miksowanie:** Strumień wideo i audio są scalane w kontenerze **MP4**.
3. **Zapis na dysku:** Gotowy plik jest zapisywany w dedykowanym katalogu, co umożliwia jego natychmiastowe udostępnienie przez serwer WWW.

## Ograniczenia implementacyjne (*RAM Management*)

Należy podkreślić, że przyjęta strategia buforowania całej sesji w RAM przed zapisem narzuca istotne **ograniczenia eksploatacyjne**. Długotrwałe nagrywanie prowadziłoby do liniowego wzrostu zużycia pamięci, grożąc jej wyczerpaniem (*błąd Out Of Memory*). Z tego względu, zaprojektowane rozwiązanie jest zoptymalizowane do rejestracji **krótkich sekwencji zdarzeń** (tzw. *klipów*), typowych dla systemów detekcji ruchu.

### 2.5.13 Konteneryzacja i wdrożenie

Zwieńczeniem procesu implementacji było przygotowanie środowiska wdrażania opartego na **konteneryzacji**. Zastosowanie technologii **Docker** pozwoliło na hermetyzację całej aplikacji wraz z jej zależnościami systemowymi, gwarantując identyczne zachowanie rozwiązania niezależnie od platformy hosta.

## Konstrukcja obrazu i zależności systemowe

Jako fundament rozwiązania wybrano obraz bazowy `python:3.13-slim`. Decyzja o użyciu wersji „slim” (zredukowanej) podyktowana była koniecznością **minimalizacji rozmiaru** wynikowego.

Istotnym wyzwaniem było zapewnienie wsparcia dla bibliotek **OpenCV** oraz **MoviePy**, które posiadają natywne zależności spoza ekosystemu Pythona. W procesie budowania obrazu zaimplementowano instalację pakietów systemowych poziomu OS:

- `ffmpeg`: Niezbędny do transkodowania audio i składania plików wideo.
- `libglib1 / libglib2.0-0`: Biblioteki graficzne wymagane przez `opencv-python-headless` do operacji na macierzach obrazu.

Zarządzanie zależnościami Python wewnętrz kontenera powierzono narzędziu `uv`, które instaluje pakiety bezpośrednio z pliku blokady `uv.lock`, zapewniając **determinizm wersji**.

## Adaptacja konfiguracji (Zmienne środowiskowe)

W celu dostosowania aplikacji do **standardów konteneryzacji**, zmodyfikowano logikę ładowania konfiguracji w module `app/settings.py`. Zrezygnowano ze sztywnego polegania na pliku `config.json` na rzecz priorytetyzacji **zmiennych środowiskowych**.

Zaimplementowany mechanizm w pierwszej kolejności sprawdza obecność zmiennych systemowych (np. TAPO\_IP) za pomocą os.environ. Dopiero w przypadku ich braku, system podejmuje próbę odczytu lokalnego pliku konfiguracyjnego.

Zmiana ta umożliwiła **bezpieczne przekazywanie poświadczeń** do kontenera w momencie jego uruchamiania (*Run-time Injection*), bez konieczności „wypalania” haseł wewnętrz obrazu Docker, co jest zgodne z najlepszymi praktykami *DevSecOps*.

## 2.6 Podsumowanie

W niniejszym rozdziale przedstawiono kompletny **proces projektowy i implementacyjny** autorskiego systemu nadzoru wizyjnego, stanowiącego otwartą alternatywę dla zamkniętego ekosystemu TP-Link. Prace rozpoczęto od przyjęcia metodyki **Double Diamond**, która pozwoliła na precyzyjne zdefiniowanie wymagań architektonicznych, a następnie na dobór optymalnego stosu technologicznego opartego na języku **Python 3.13**, bibliotece **OpenCV** oraz **konteneryzacji Docker**.

Kluczowym osiągnięciem inżynierskim opisany w tej części pracy jest **skuteczna integracja warstwy sprzętowej kamery Tapo C200 z aplikacją webową**, pomimo ograniczeń narzuconych przez producenta (zjawisko *vendor lock-in*). Zrealizowano to poprzez zaprojektowanie **trójwarstwowej architektury systemu**, w której:

- **Warstwa Abstrakcji Sprzętu (HAL)** skutecznie izoluje logikę aplikacji od specyfiki protokołów *RTSP* i własnościowego *API* sterującego, wykorzystując **inżynierię wsteczną** do obsługi funkcji *PTZ*.
- **Warstwa Logiki (Middleware)** realizuje zaawansowane przetwarzanie sygnałów w czasie rzeczywistym.
  - Dzięki zastosowaniu architektury potokowej (*Pipe and Filter*) oraz hybrydowemu podejściu do obsługi multimedialnych (*OpenCV* dla wideo, *PyAV* dla audio), rozwiązano problemy **synchronizacji A/V** oraz **kompensacji opóźnień sieciowych (Jitter Buffer)**.
  - Zaimplementowano również autorski algorytm **detekcji ruchu** działający na **brzegu sieci** (*Edge Computing*), uniezależniając system od chmury obliczeniowej.
- **Warstwa Prezentacji** zapewnia interaktywność i niski czas reakcji dzięki wykorzystaniu protokołu **WebSocket** oraz renderowaniu obrazu na elemencie **HTML5 Canvas**, co eliminuje narzut typowy dla klasycznych odtwarzaczy wideo.

Całość rozwiązania została **zhermetyzowana w kontenerze Docker**, co gwarantuje powtarzalność środowiska uruchomieniowego i łatwość wdrożenia. Opisany proces implementacji doprowadził do powstania funkcjonalnego artefaktu programistycz-

nego, gotowego do empirycznej weryfikacji. Kolejny rozdział poświęcony zostanie **testom wydajnościowym** oraz analizie jakościowej tak przygotowanego rozwiązania.

### 3 Testowanie i analiza wyników

Niniejszy rozdział stanowi kluczowy etap weryfikacji opracowanego rozwiązania programistycznego. Po fazie projektowania i implementacji, niezbędne jest poddanie systemu rygorystycznym testom, które pozwolą ocenić stopień realizacji postawionych celów inżynierskich. Głównym założeniem niniejszej części pracy jest nie tylko potwierdzenie poprawności działania poszczególnych modułów, ale przede wszystkim uzyskanie pogłębionej wiedzy na temat charakterystyki operacyjnej systemu w warunkach rzeczywistych. Proces testowania został zaprojektowany tak, aby zidentyfikować potencjalne ograniczenia wydajnościowe oraz słabe punkty tzw. „**wąskie gardła**” zbudowanego systemu. W systemach **Internetu Rzeczy (IoT)** przetwarzających multimedia w czasie rzeczywistym, krytyczne znaczenie ma balans pomiędzy jakością obrazu, opóźnieniem transmisji (**latency**) a zużyciem zasobów sprzętowych hosta. Analiza uzyskanych wyników pozwoli na sformułowanie konkretnych wniosków optymalizacyjnych, które mogą posłużyć jako fundament dla przyszłego rozwoju oprogramowania, dążącego do zwiększenia jego skalowalności i odporności na błędy.

W ramach przeprowadzonych badań zweryfikowano następujące obszary funkcjonalne:

1. **Stabilność i płynność strumieniowania:** weryfikacja transmisji video.
2. **Efektywność algorytmu detekcji ruchu:** analiza procesu realizowanego na brzegu sieci (**edge processing**).
3. **Wpływ rejestracji na zasoby:** badanie obciążenia pamięci operacyjnej systemu podczas zapisu materiału wideo.

Poprzez kwantyfikację tych parametrów, możliwe będzie obiektywne stwierdzenie, w jakim stopniu autorskie rozwiązanie oparte na oprogramowaniu **Open Source** stanowi skutecną i bezpieczną alternatywę dla zamkniętego ekosystemu producenta.

#### 3.1 Środowisko testowe

W celu weryfikacji wydajności potoków multimedialnych oraz stabilności sterowania PTZ, przygotowano dedykowane środowisko badawcze. System uruchomiono na stacji roboczej pełniącej rolę bramy IoT (Gateway), komunikującej się z kamerą wewnętrz odizolowanej sieci lokalnej.

##### **Infrastruktura sprzętowa i systemowa (Host)**

Głównym węzłem obliczeniowym, na którym uruchomiono konteneryzowaną aplikację, był komputer o następującej specyfikacji:

Tabela 3.1: Specyfikacja techniczna stacji roboczej (Host).

Komponent	Parametry
Procesor (CPU)	11th Gen Intel Core i7-11370H @ 4.8 GHz
Pamięć RAM	15.37 GiB
Pamięć masowa	474.92 GiB (system plików btrfs)
System operacyjny	Arch Linux (Kernel 6.17.9-arch1-1)
Architektura	x86_64

## Urządzenie końcowe i runtime

Do testów wykorzystano kamerę **TP-Link Tapo C200** (Firmware 1.3.1) skonfigurowaną w rozdzielcości **Full HD (1080p)** przy 30 FPS. Rozwiążanie zostało w pełni odizolowane od systemu operacyjnego poprzez stos technologiczny:

- **Konteneryzacja:** Docker Engine (izolacja procesów).
- **Interpreter:** Python 3.13.
- **Zarządzanie pakietami:** Narzędzie *uv* zapewniające determinizm bibliotek.

## Parametry sieciowe

Testy przeprowadzono w stabilnej sieci bezprzewodowej LAN, aby zminimalizować błędy transmisji:

- **Siła sygnału:** -45 dBm (bardzo dobra).
- **Opóźnienie (RTT):** Średnio 3 ms do urządzenia.

## 3.2 Przebieg scenariuszy testowych

Poniżej przedstawiono szczegółowy opis procedur badawczych oraz uzyskane parametry techniczne dla poszczególnych modułów systemu.

### 3.2.1 Test T01: Analiza wydajności algorytmu detekcji ruchu

Celem pierwszego scenariusza testowego była weryfikacja charakterystyki wydajnościowej modułu MotionDetector. Badanie miało dowieść, czy zaprojektowany algorytm analizy obrazu jest w stanie pracować w czasie rzeczywistym przy pełnej rozdzielcości sensora kamery Tapo C200.

**Metodologia i przebieg badania** Test został przeprowadzony w kontrolowanym środowisku wykonawczym przy użyciu dedykowanego skryptu benchmarkowego. Procedura badawcza obejmowała następujące etapy:

1. **Inicjalizacja środowiska:** Detektor ruchu został skonfigurowany z progiem czułości `min_area=1000`, co odpowiada założeniom projektowym minimalizacji fałszywych alarmów.
2. **Faza stabilizacji:** Wykonano 10 iteracji rozgrzewkowych w celu ustabilizowania zasobów procesora oraz załadowania bibliotek OpenCV do pamięci podręcznej.
3. **Generowanie obciążenia:** Przeprowadzono 500 iteracji testowych na klatce o rozdzielcości **1080p** (1920x1080 px).
4. **Symulacja warunków rzeczywistych:** Do każdej klatki dodawano losowy szum cyfrowy (zakres 0–50) przy użyciu funkcji `cv2.add`, aby wymusić pełną ścieżkę obliczeniową algorytmu.

**Jednostki i wyniki parametrów pomiarowych** W trakcie testu monitorowano dwa kluczowe wskaźniki inżynierskie:

- **Średni czas przetwarzania klatki ( $t_{avg}$ ):** Wyrażony w milisekundach (ms), określa czas pełnego potoku analizy.
  - Wynik: **65,1 ms.**
- **Teoretyczna maksymalna wydajność ( $FPS_{theor}$ ):** Obliczana jako odwrotność średniego czasu przetwarzania:

$$FPS_{theor} = \frac{1}{t_{avg}}$$

- Wynik:  **$\approx 15,36 \text{ FPS.}$**

### 3.2.2 Test T02: Stabilność i płynność strumieniowania video

Drugi scenariusz testowy koncentrował się na weryfikacji jakości transmisji obrazu w czasie rzeczywistym. Badanie zostało podzielone na dwa odrębne etapy: ocenę stabilności liczby wyświetlanych klatek na sekundę oraz pomiar całkowitego opóźnienia przesyłu danych od sensora do interfejsu użytkownika.

## **Podtest A: Wydajność klatkowa i stabilność (FPS)**

Celem badania było określenie końcowej przepustowości wizualnej systemu, uwzględniającej pełny potok przetwarzania: od przechwycenia strumienia RTSP, poprzez dekodowanie i skalowanie na serwerze, aż po renderowanie na płótnie HTML5 Canvas w przeglądarce klienta.

**Metodologia i przebieg badania** Procedura testowa wymagała modyfikacji kodu źródłowego aplikacji w celu zaimplementowania mechanizmów logowania parametrów czasowych każdej wrenderowanej klatki.

1. Aplikacja została poddana trzem seriom pomiarowym, z których każda trwała 30 sekund.
2. Pomiędzy seriami następował restart systemu, co pozwoliło na wyeliminowanie wpływu ewentualnej fragmentacji pamięci lub przepełnienia buforów na wyniki końcowe.
3. Łącznie zgromadzono **347 próbek** danych pomiarowych.

**Mierzone parametry i wyniki** W trakcie testu monitorowano chwilową wartość klatek na sekundę (FPS). Uzyskane dane statystyczne przedstawiają się następująco:

- **Średnia wartość FPS:** 15,70.
- **Odchylenie standardowe:** 1,57 FPS.
- **Wartości krytyczne:** Odnotowano sporadyczne spadki płynności do poziomu 10 FPS.

W trakcie monitorowania logów systemowych zaobserwowano, że spadki wydajności występowali synchronicznie z błędami zgłaszanymi przez dekoder strumienia H.264 (np. error while decoding MB), co wskazuje na okresowe problemy z integralnością danych w warstwie transportowej.

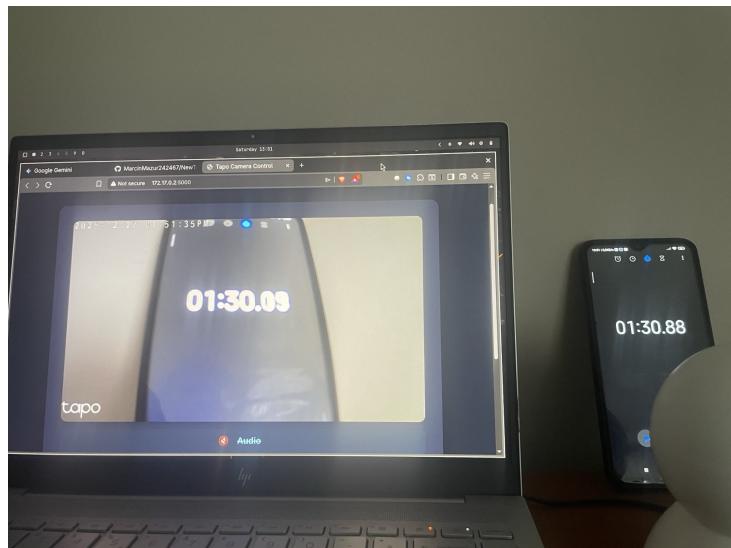
## **Podtest B: Opóźnienie przesyłu (End-to-End Latency)**

Badanie opóźnienia miało na celu określenie całkowitego czasu potrzebnego na przejście informacji wizualnej przez wszystkie warstwy architektury rozwiązania.

**Metodologia i przebieg badania** Do realizacji pomiaru wykorzystano metodę porównawczą z użyciem wzorcowego źródła czasu.

1. Kamera została skierowana na ekran monitora (Ekran nr 1), na którym uruchomiony został stoper cyfrowy o wysokiej precyzji.

2. Interfejs webowy projektowanej aplikacji wyświetlany był na drugim monitorze (Ekran nr 2).
3. Test polegał na wykonaniu serii zdjęć obu ekranów w tym samym momencie.
4. Wartość opóźnienia wyliczano jako bezwzględną różnicę pomiędzy czasem wyświetlanym na stoperze źródłowym a czasem widocznym na podglądzie w aplikacji.



Rysunek 3.1: Fotografia przedstawiająca przebieg badania.

**Mierzone parametry i wyniki** Na podstawie 11 prób kontrolnych wyznaczono charakterystykę opóźnienia systemu:

- **Średnie opóźnienie (Mean Latency):** 728 ms.
- **Wartość minimalna:** 610 ms.
- **Wartość maksymalna:** 800 ms.
- **Zmienna opóźnienia (Jitter):** 57 ms.

Uzyskane wyniki pozwalają na ocenę responsywności systemu, co jest kluczowe w kontekście zdalnego sterowania mechaniką PTZ kamery oraz interakcji użytkownika z systemem.

### 3.2.3 Test T03: Wpływ procesu rejestracji na zasoby

Ostatni etap testów koncentrował się na analizie obciążenia pamięci RAM podczas procesu zapisu materiału wideo. Badanie to miało na celu zweryfikowanie skalowalności przyjętej architektury nagrywania, opartej na strategii odroczonego zapisu (ang. *Deferred Writing*).

**Metodologia i cel badania** Celem testu było określenie wpływu mechanizmu buforowania klatek w pamięci operacyjnej na stabilność systemu. W zaimplementowanym rozwiążaniu, podczas trwania nagrania, surowe dane wizyjne i foniczne są gromadzone w listach systemowych (`self.video_frames`, `self.audio_chunks`). Operacja zapisu na dysk następuje dopiero po zakończeniu sesji nagrywania. Pomiary przeprowadzono podczas ciągłej sesji rejestracji strumienia o rozdzielczości Full HD.

**Wyniki pomiarów** W trakcie badania zaobserwowano bezpośrednią zależność między czasem trwania sesji a zajętością zasobów. Wyniki testu T03 wykazały:

- **Charakterystyka wzrostu:** Zaobserwowano liniowy przyrost zużycia pamięci RAM.
- **Tempo wzrostu:** System rezerwował średnio 148,3 MB na każdą sekundę nagranego materiału.

Zastosowany model matematyczny wzrostu można opisać wzorem:

$$M(t) = a \cdot t \quad (1)$$

Gdzie:

- $M$  – całkowita zajętość pamięci przez bufor (MB),
- $t$  – czas trwania nagrania w sekundach,
- $a \approx 148,3 \text{ MB/s}$  – współczynnik przyrostu wynikający z rozmiaru nieskompresowanych klatek w pamięci.

### 3.3 Wnioski i analiza

#### 3.3.1 Algorytm detekcji ruchu

Na podstawie przeprowadzonych pomiarów wydajnościowych sformułowano następujące wnioski dotyczące działania modułu detekcji:

**Potwierdzenie pracy w czasie rzeczywistym** Wyniki testów wykazały, że system osiągnął średnią wydajność przetwarzania na poziomie **15,36 FPS**. Wartość ta przewyższa nominalną prędkość nadawania strumienia przez kamerę, wynoszącą 15 FPS. Oznacza to, że nadzędny cel inżynierski został spełniony – zaimplementowany algorytm jest w stanie przetwarzać obraz na bieżąco (ang. *on-the-fly*), nie wprowadzając opóźnień w procesie przesyłu wideo, co jest kluczowe dla systemów monitoringu.

**Wysoki koszt obliczeniowy dla rozdzielczości Full HD** Zauważono istotne obciążenie zasobów przy pracy z obrazem o wysokiej rozdzielczości. Mimo wykorzystania wydajnej jednostki centralnej (Intel Core i7), zarejestrowany zapas mocy obliczeniowej jest minimalny i wynosi zaledwie **0,36 FPS** powyżej wymaganego progu płynności. Wskazuje to jednoznacznie, że proces analizy każdej pojedynczej klatki w pełnej rozdzielczości 1920x1080 pikseli jest operacją wysoce wymagającą dla skryptu realizowanego w języku Python.

**Skuteczność symulacji obciążenia granicznego** Zastosowana metodyka testowa polegająca na wprowadzaniu sztucznego szumu cyfrowego (z wykorzystaniem funkcji cv2.add) okazała się skuteczna. Zabieg ten wymusił na algorytmie pracę w najtrudniejszych warunkach obliczeniowych. Uzyskany wynik testu można zatem uznać za miarodajny dla scenariusza typu „najgorszy przypadek” (ang. *worst-case scenario*), obejmującego nagłe zmiany oświetlenia lub wystąpienie silnych zakłóceń obrazu.

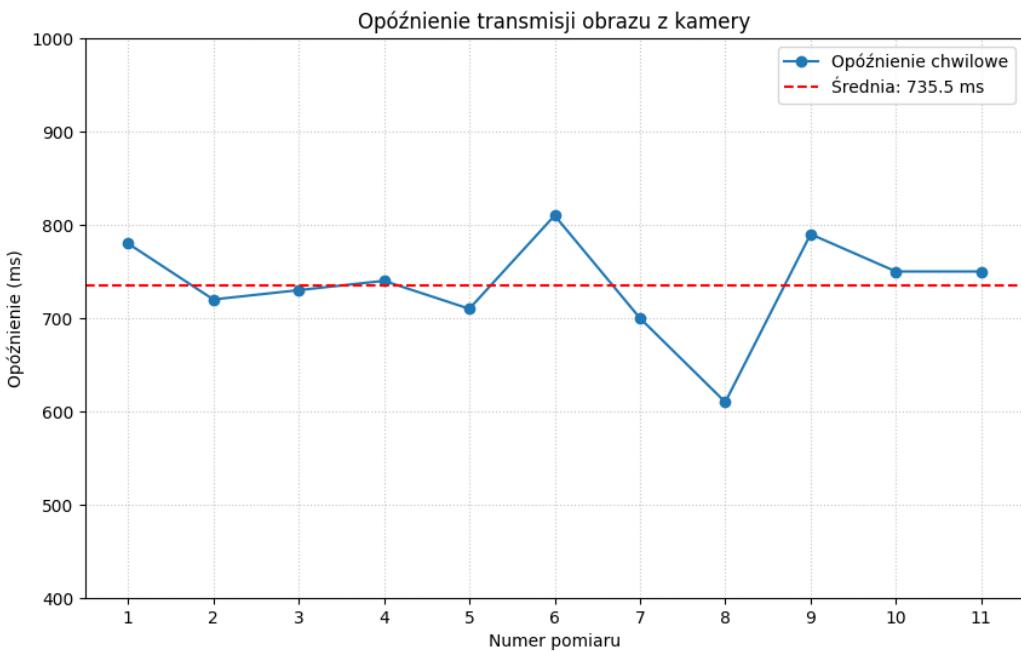
**Stabilność środowiska konteneryzacji** Badanie potwierdziło stabilność działania aplikacji w środowisku wirtualizowanym. Uruchomienie systemu wewnętrz kontenera Docker nie wpłynęło negatywnie na jego niezawodność. Brak błędów wykonawczych (ang. *Runtime Errors*) podczas próby obejmującej 500 iteracji stanowi dowód na to, że przydzielone zasoby są wystarczające do zapewnienia ciągłej pracy detektora w izolowanym środowisku.

**Narzut interpretacyjny języka Python** Zidentyfikowano ograniczenia wydajnościowe wynikające z wybranego języka programowania. W celu zwiększenia efektywności algorytmu i uzyskania większego marginesu bezpieczeństwa FPS, rekomendowane jest przeniesienie obliczeń macierzowych na procesor graficzny (GPU/CUDA) lub przepisanie krytycznych sekcji kodu (ang. *hot paths*) do języka komplikowanego, takiego jak C++.

### 3.3.2 Stabilność i płynność strumieniowania

Na podstawie danych zgromadzonych podczas testu strumieniowania, sformułowano wnioski dotyczące trzech kluczowych aspektów działania systemu: wydajności przetwarzania, opóźnień transmisji oraz stabilności obrazu.

**Pełna przepustowość przetwarzania (Wydajność FPS)** Pomiary wykazały, że system osiągnął średnią prędkość odświeżania na poziomie **15,7 FPS** przy źródle nadającym nominalnie 15 FPS. Oznacza to, że aplikacja skutecznie przetwarza 100% dostarczonego materiału wideo, a niewielka nadwyżka wynika z różnic w taktowaniu zegarów systemowych. Badanie potwierdziło brak występowania zjawiska „wąskiego



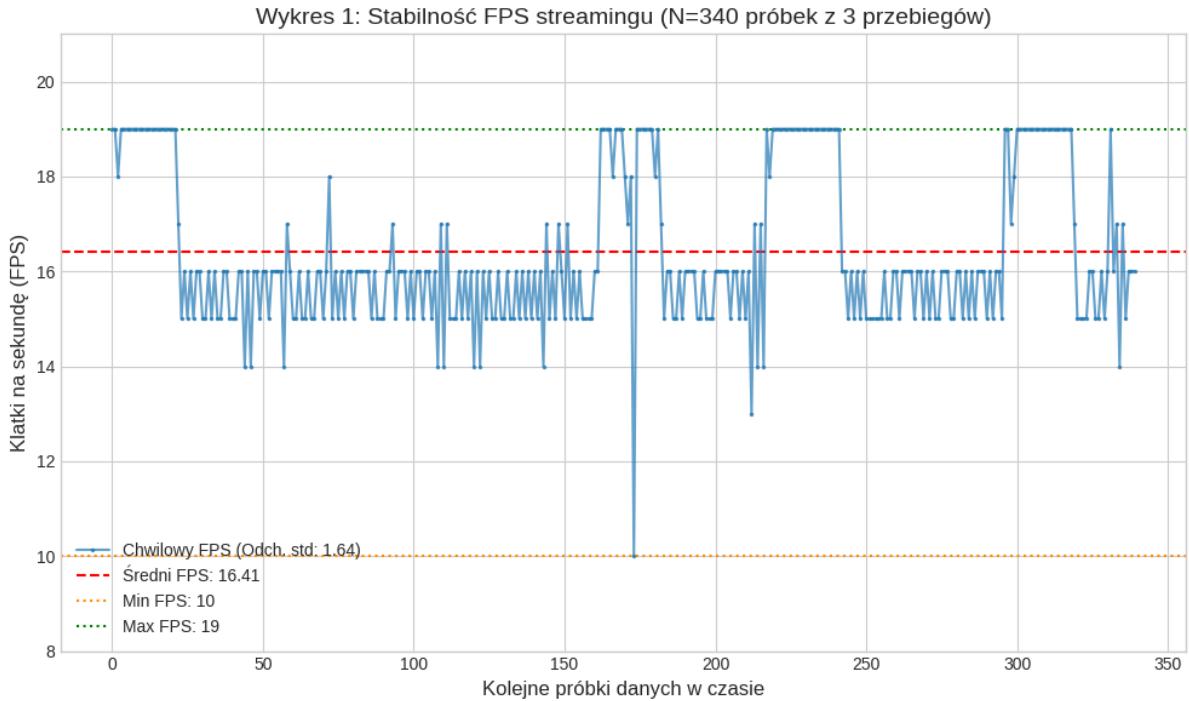
Rysunek 3.2: Wykres opóźnienia end-to-end podczas testu T02.

gardła” (ang. *bottleneck*) zarówno po stronie serwera aplikacyjnego, jak i klienta webowego, co świadczy o efektywnej implementacji potoku *Video Pipeline*.

**Nadmiarowość sprzętowa i skalowalność** Analiza obciążenia wskazuje, że wykorzystanie procesora klasy Intel Core i7 do obsługi pojedynczego strumienia 1080p stanowi rozwiązanie z bardzo dużym zapasem mocy obliczeniowej. System wykazuje potencjał do skalowania wertykalnego – teoretycznie jest zdolny do równoległej obsługi wielu kamer bez degradacji płynności obrazu, co jest istotnym atutem w kontekście rozbudowy instalacji monitoringu.

**Charakterystyka opóźnienia (Latency)** Zmierzone średnie opóźnienie typu *end-to-end* na poziomie **728 ms** zidentyfikowano jako rezultat przyjętej architektury programowej, a nie braku zasobów sprzętowych. Głównymi czynnikami wpływającymi na ten wynik są mechanizmy wewnętrzne buforowania biblioteki OpenCV oraz narzut komunikacyjny protokołu WebSocket. Z perspektywy użyteczności, wartość ta jest w pełni akceptowalna dla zastosowań monitoringu pasywnego (obserwacji). Należy jednak odnotować, że podczas aktywnego sterowania mechaniką PTZ, opóźnienie bliskie jednej sekundy może być odczuwalne dla operatora, wpływając na precyzję manualnego śledzenia obiektów.

**Stabilność transmisji i wrażliwość kodnika H.264** Zaobserwowane podczas testów artefakty wizualne (błędy dekodowania) potwierdziły specyfikę pracy z protokołem UDP



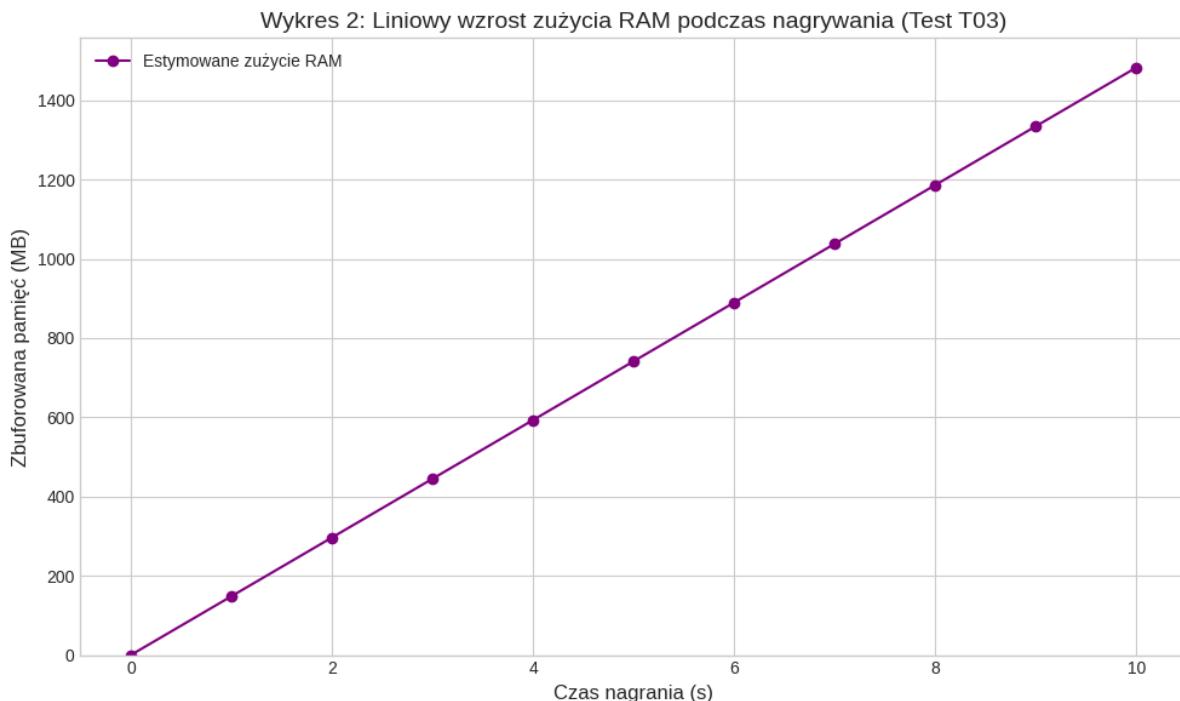
Rysunek 3.3: Wykres stabilności FPS podczas testu T02.

w środowisku sieci bezprzewodowych. Choć protokół ten zapewnia szybszą transmisję niż TCP, brak mechanizmu retransmisji pakietów w połączeniu z charakterystyką kodnika H.264 (wysoka kompresja międzyklatkowa) sprawia, że nawet minimalna utrata danych w sieci Wi-Fi skutkuje widocznymi błędami w obrazie. Jest to akceptowalny kompromis projektowy na rzecz utrzymania charakterystyki czasu rzeczywistego.

### 3.3.3 Analiza wpływu procesu rejestracji na zasoby

Test obciążeniowy pamięci operacyjnej (T03) dostarczył kluczowych danych dotyczących skalowalności modułu rejestracji (Recorder). Na podstawie zaobserwowanej charakterystyki zużycia zasobów sformułowano następujące wnioski krytyczne:

**Dyskwalifikacja metody „Odroczonego Zapisu” w zastosowaniach ciągłych** Za stosowana strategia buforowania całego materiału wideo w pamięci RAM (ang. *In-Memory Buffering*), opisana w sekcji 3.5.11, okazała się nieprzydatna w warunkach produkcyjnych wymagających ciągłości działania. Przy odnotowanym tempie konsumpcji pamięci na poziomie **148,3 MB/s**, rozwiązanie to jest ograniczone funkcjonalnie wyłącznie do rejestracji bardzo krótkich sekwencji zdarzeń (ang. *Short Clips*), takich jak kilkusekundowe klipy z detekcji ruchu. Metoda ta nie może być stosowana do monitoringu ciągłego (24/7).



Rysunek 3.4: Wykres zużycia pamięci RAM podczas testu T03.

**Istotny narzut technologiczny środowiska uruchomieniowego (Overhead)** Analiza porównawcza wykazała znaczącą nieefektywność strukturalną wybranego stosu technologicznego w kontekście przechowywania dużych wolumenów danych binarnych.

- **Teoretyczny strumień danych:**  $\approx 93 \text{ MB/s}$  (dla nieskompresowanych klatek 1080p).
- **Rzeczywiste zużycie:**  $\approx 148 \text{ MB/s}$ .

Różnica ta wskazuje na ok. **59% narzut pamięciowy**, wynikający z narzutu obiektowego języka Python oraz sposobu alokacji pamięci dla list przechowujących obiekty biblioteki NumPy. Oznacza to, że środowisko uruchomieniowe zużywa ponad połowę alokowanych zasobów na obsługę samej struktury danych, a nie na użyteczną treść wideo.

**Krytyczne ograniczenie czasu nagrywania (Time-to-Crash)** Mimo dysponowania stacją roboczą wyposażoną w ponad 15 GB pamięci RAM, stabilność systemu jest gwarantowana jedynie przez okres około **1 minutę i 40 sekund**. Przekroczenie tego czasu prowadzi do całkowitego wyczerpania dostępnej pamięci i awarii krytycznej aplikacji (ang. *Crash*). Ekstrapolując te wyniki na standardowe urządzenia brzegowe IoT, takie jak Raspberry Pi (zazwyczaj 4 GB RAM), bezpieczny czas nagrywania uległby skróceniu do zaledwie  $\approx 25$  sekund, co drastycznie ogranicza użyteczność systemu na docelowej platformie sprzętowej.

**Brak kompresji w czasie rzeczywistym jako przyczyna saturacji** Zidentyfikowano główną przyczynę problemu wydajnościowego, którą jest przechowywanie w pamięci „surowych” klatek obrazu (bitmap w formacie BGR/RGB). Brak implementacji kompresji strumieniowej (np. potokowego kodowania do H.264 w locie) sprawia, że dane buforowane w RAM zajmują setki razy więcej miejsca niż wynikowy, skompresowany plik wideo zapisywany ostatecznie na dysku. Wskazuje to na konieczność zmiany architektury modułu Recorder w przyszłych iteracjach projektu, np. poprzez bezpośrednie przekazywanie strumienia do procesu kodującego (ang. *FFmpeg pipe*).

### 3.4 Synteza wniosków

Przeprowadzone w niniejszym rozdziale badania wydajnościowe oraz testy weryfikacyjne (T01–T03) pozwoliły na empiryczną ocenę stopnia realizacji celu głównego pracy, jakim było stworzenie niezależnego systemu monitoringu opartego na rozwiązańach **Open Source**. Analiza wyników umożliwiła również identyfikację kluczowych ograniczeń technologicznych zaprojektowanego rozwiązania.

#### 1. Stopień realizacji celu głównego i celów szczegółowych

Należy uznać, że cel inżynierski został osiągnięty. Zbudowano i wdrożono funkcjonalny, skonteneryzowany system, który skutecznie uniezależnia użytkownika od infrastruktury chmurowej producenta, rozwiązuje problem *vendor lock-in*.

- **Płynność obrazu:** System osiągnął pełną wydajność przetwarzania (średnio 15,7 FPS przy nadawaniu 15 FPS), co potwierdza weryfikację pozytywną założenia o możliwości pracy w czasie rzeczywistym.
- **Skalowalność i stabilność:** Środowisko kontenerowe Docker wykazało pełną stabilność operacyjną, nie generując mierzalnych błędów narzutowych, co spełnia postulat dotyczący modułowości i przenośności systemu.

#### 2. Ograniczenia wydajnościowe (CPU i narzut środowiska Python)

Analiza wyników testu T01 na platformie referencyjnej (Intel Core i7-11370H) wykazała, że interpretowany język Python stanowi istotne wąskie gardło (ang. *bottleneck*) dla przetwarzania obrazu o wysokiej rozdzielczości (Full HD  $1920 \times 1080$  px). Mimo wykorzystania jednostki CPU o wysokim taktowaniu (do 4.8 GHz), zarejestrowany zapas mocy obliczeniowej był marginalny (zaledwie 0,36 FPS powyżej progu płynności). Obserwacja ta potwierdza słuszność decyzji projektowej o wyłączeniu z zakresu pracy implementacji zaawansowanych modeli głębokiego uczenia (np. YOLO). Skoro podstawowa detekcja różnicowa utylizuje niemal pełne zasoby wątku procesora, wdrożenie złożonej analityki AI wymagałoby zastosowania akceleracji sprzętowej (GPU) lub migracji krytycznych sekcji kodu do języka komplikowanego (C++).

### **3. Wpływ opóźnień na sterowanie mechaniką PTZ**

Weryfikacja w warunkach sieci bezprzewodowej (Test T02) wykazała średnie opóźnienie typu *end-to-end* na poziomie 728 ms. Jest to wartość w pełni akceptowalna dla zastosowań monitoringu pasywnego. Jednakże, w kontekście aktywnego sterowania kamerą (PTZ), opóźnienie to staje się odczuwalne dla operatora i obniża precyzję manualnego pozycjonowania głowicy. Należy podkreślić, że latencja ta wynika głównie z mechanizmów buforowania biblioteki OpenCV oraz narzutu protokołów komunikacyjnych, a nie z niedostatków mocy obliczeniowej.

### **4. Krytyczna ocena strategii zapisu wideo**

Test T03 negatywnie zweryfikował przyjętą strategię „Odroczonego Zapisu”, polegającą na buforowaniu surowych klatek w pamięci RAM.

- **Nieefektywność alokacji:** Przechowywanie nieskompresowanych obiektów tablicowych generuje zużycie pamięci rzędu 148 MB/s.
- **Awaryjność:** Nawet przy dyspozycji stacji roboczej z 16 GB pamięci RAM, system ulega awarii krytycznej po ok. 100 sekundach nagrania.
- **Wniosek inżynierski:** W systemach wbudowanych klasy IoT niezbędna jest implementacja kompresji strumieniowej w czasie rzeczywistym (np. kodowanie danych bezpośrednio do enkodera H.264), a rezygnacja z buforowania surowych bitmap jest warunkiem koniecznym dla stabilności procesu rejestracji.

### **5. Stabilność transmisji w środowisku bezprzewodowym**

Wykorzystanie protokołu UDP do transmisji wideo w sieci Wi-Fi (przy sile sygnału -45 dBm) zapewniło pożądane niskie opóźnienia, odbyło się to jednak kosztem integralności wizualnej obrazu. Wysoka wrażliwość kodeka H.264 na utratę pakietów skutkowała okresowym pojawianiem się artefaktów. Dla środowisk produkcyjnych rekomendowane jest zastosowanie połączenia przewodowego (Ethernet) lub wdrożenie programowego bufora korekcyjnego (ang. *jitter buffer*), co jednak wiążałoby się z kompromisem w postaci zwiększonego opóźnienia.

## **Wnioski Końcowe**

Przeprowadzona analiza teoretyczna, proces implementacji oraz weryfikacja empiryczna prototypu systemu pozwalają na sformułowanie ostatecznych wniosków dotyczących stopnia realizacji założeń pracy. Projekt miał na celu rozwiązanie istotnego problemu inżynierskiego, jakim jest uzależnienie funkcjonalności sprzętu IoT od eko-systemu producenta (ang. *vendor lock-in*), na przykładzie kamery TP-Link Tapo C200.

### **Stopień realizacji celu głównego**

Cel główny pracy, zdefiniowany jako opracowanie kompletnego, modułowego rozwiązania programistycznego opartego na otwartym oprogramowaniu (**Open Source**), umożliwiającego uniezależnienie kamery od chmury producenta, został osiągnięty w stopniu pełnym.

Zbudowany system, wykorzystujący język Python, bibliotekę OpenCV oraz konteneryzację Docker, skutecznie przejmuje kontrolę nad strumieniami audio-video oraz mechaniką kamery w izolowanej sieci lokalnej. Udowodniono, że zastosowanie inżynierii wstępnej oraz otwartych standardów pozwala na bezpieczną eksploatację konsumenckich urządzeń IoT z pominięciem dedykowanych aplikacji mobilnych i infrastruktury zewnętrznej.

### **Analiza realizacji celów szczegółowych**

Weryfikacja celów szczegółowych, oparta na wynikach testów T01–T03 (opisanych w Rozdziale 4), przedstawia się następująco:

#### **1. Stabilne wyświetlanie obrazu w czasie rzeczywistym**

**Status:** Cel osiągnięty.

**Uzasadnienie:** Pomiary wykazały średnią płynność na poziomie **15,7 FPS**, co w pełni pokrywa nominalną prędkość nadawania kamery (15 FPS). System nie gubi klatek w procesie przetwarzania. Zmierzone opóźnienie (ang. *latency*) rzędu 728 ms jest wartością akceptowaną dla systemów nadzoru wizyjnego realizowanych w technologiach webowych, choć stanowi pewne wyzwanie przy precyzyjnym sterowaniu manualnym.

#### **2. Sterowanie funkcjami PTZ (Pan/Tilt/Zoom)**

**Status:** Cel osiągnięty.

**Uzasadnienie:** Poprzez implementację warstwy abstrakcji sprzętowej (HAL) i wykorzystanie biblioteki PyTapo, udało się skutecznie zemulować szyfrowaną komunikację z API kamery. System umożliwia pełne sterowanie silnikami krokowymi

urządzenia z poziomu przeglądarki, co stanowi funkcjonalność niedostępną w standardowym protokole ONVIF dla tego modelu kamery.

### 3. Implementacja algorytmu wykrywania ruchu na serwerze

**Status:** Cel osiągnięty.

**Uzasadnienie:** Zaimplementowany algorytm adaptacyjnego modelowania tła (ang. *Background Subtraction*) skutecznie identyfikuje zdarzenia w czasie rzeczywistym. Testy wydajnościowe (T01) potwierdziły, że system jest w stanie przetwarzać obraz Full HD z prędkością **15,36 FPS**. Należy jednak odnotować, że proces ten generuje wysokie obciążenie jednostki CPU, co wskazuje na narzut interpretowanego języka Python przy operacjach na dużych macierzach danych.

### 4. Skalowalność i reprodukowalność (Docker)

**Status:** Cel osiągnięty.

**Uzasadnienie:** Zastosowanie technologii Docker oraz menedżera pakietów uv zapewniło pełną izolację środowiska uruchomieniowego. Testy potwierdziły stabilność działania aplikacji w kontenerze, eliminując problemy z konfliktami bibliotek systemowych (np. libgl1, ffmpeg). Architektura ta umożliwia łatwe wdrożenie rozwiązania na platformach wbudowanych.

### 5. Zapis nagrań wideo i odtwarzanie

**Status:** Cel osiągnięty z ograniczeniami eksploracyjnymi.

**Uzasadnienie:** Funkcjonalność zapisu i odtwarzania została zaimplementowana i działa poprawnie – system generuje pliki MP4 z synchronizacją audio/video, które są dostępne w interfejsie webowym. Jednakże, analiza testu T03 ujawniła krytyczną wadę architektoniczną w postaci strategii buforowania w pamięci RAM (ang. *Deferred Writing*). Przy zużyciu pamięci na poziomie  $\approx 148$  MB/s, rozwiązanie w obecnym kształcie nadaje się wyłącznie do rejestracji krótkich zdarzeń (tzw. klipów alarmowych), a nie do ciągłego monitoringu 24/7. Jest to obszar wymagający optymalizacji w przyszłych wersjach oprogramowania.

Niniejsza praca udowadnia, że możliwe jest zbudowanie profesjonalnej bramy IoT (ang. *Gateway*) dla tanich kamer konsumenckich przy użyciu wyłącznie technologii **Open Source**. Zaprojektowana architektura trójwarstwowa (Prezentacja – Logika – Sprzęt) zdała egzamin, zapewniając separację logiki biznesowej od specyfiki sprzętowej.

Głównym wnioskiem płynącym z realizacji pracy jest fakt, że „uwolnienie” sprzętu od producenta wiąże się z koniecznością przejęcia przez programistę odpowiedzialności za optymalizację niskopoziomową. Wybór wysokopoziomowych narzędzi (Python/OpenCV), choć przyspieszył proces prototypowania (ang. *Rapid Prototyping*), ujawnił

swoje ograniczenia wydajnościowe przy przetwarzaniu strumieni HD w czasie rzeczywistym. Mimo to, stworzony system stanowi funkcjonalną, bezpieczną i prywatną alternatywę dla rozwiązań chmurowych.

## Kierunki dalszego rozwoju

Zrealizowany w ramach niniejszej pracy projekt inżynierski stanowi funkcjonalny prototyp (ang. *Proof of Concept*), który skutecznie demonstruje możliwość przełamania bariery *vendor lock-in* w konsumenckich kamerach IoT. Przeprowadzona w Rozdziale 4 analiza wydajnościowa zidentyfikowała jednak szereg ograniczeń architektury, które wyznaczają ścieżkę dalszej ewolucji oprogramowania. Rozwój systemu powinien koncentrować się na optymalizacji zarządzania zasobami, skalowalności oraz implementacji zaawansowanych mechanizmów analitycznych i bezpieczeństwa.

## Optymalizacja podsystemu rejestracji: Przejście na Zapis Strumieniowy

Najpilniejszym wyzwaniem inżynierskim, wynikającym bezpośrednio z krytycznych wniosków testu T03 (saturacja pamięci RAM), jest fundamentalna przebudowa modułu Recorder. Obecna strategia *Deferred Writing* (odroczonego zapisu), polegająca na buforowaniu pełnych sesji nagraniowych w pamięci operacyjnej, jest nieakceptowalna w środowisku produkcyjnym.

Rozwiązaniem docelowym jest implementacja strategii Zapisu Strumieniowego (ang. *Stream Writing*). Nowa architektura modułu powinna opierać się na wzorcu projektowym Producent-Konsument (ang. *Producer-Consumer Pattern*):

- **Bufor cykliczny (FIFO):** Pamięć RAM powinna służyć jedynie jako tymczasowy bufor (kolejka) dla kilkunastu klatek oczekujących na przetworzenie, co ustabilizuje zużycie pamięci na stałym, niskim poziomie, niezależnie od długości nagrania.
- **Asynchroniczny zapis:** Dedykowany wątek zapisu (Konsument) powinien na bieżąco pobierać klatki z kolejki i przekazywać je do enkodera (np. klasy cv2.VideoCapture w OpenCV lub potoku FFmpeg).
- **Kompresja w locie:** Kluczowe jest, aby dane trafiające na dysk były natychmiast kompresowane (np. kodekiem H.264), co wyeliminuje problem przechowywania surowych bitmap.

Taka zmiana architektury pozwoli na przekształcenie systemu z narzędzia do rejestracji krótkich klipów w pełnowartościowy rejestrator NVR (ang. *Network Video Recorder*), zdolny do pracy w trybie ciągłym 24/7.

## Implementacja warstwy persystencji danych

Obecna wersja systemu przechowuje stan aplikacji oraz konfigurację w sposób ulotny lub w prostych plikach płaskich (JSON). W celu podniesienia niezawodności i możliwości analitycznych, konieczne jest wdrożenie relacyjnej bazy danych (np. SQLite dla małych wdrożeń lub PostgreSQL dla systemów rozproszonych). Baza danych umożliwia:

- Trwałe przechowywanie logów zdarzeń (detekcja ruchu, błędy połączenia).
- Zarządzanie metadanymi nagrani video (indeksowanie po czasie, typie zdarzenia), co przyspieszy ich wyszukiwanie.
- Przechowywanie konfiguracji użytkownika oraz profili ustawień dla różnych scenariuszy monitoringu.

## Skalowalność: Obsługa wielu kamer

Architektura obecnego rozwiązania opiera się na wzorcu Singleton dla obiektu kamery, co ogranicza system do obsługi jednego urządzenia. Rozwój w kierunku obsługi wielu strumieni video wymaga refaktoryzacji warstwy *Middleware*. Należy wprowadzić dynamiczne zarządzanieinstancjami klasy `VideoStreamer`, gdzie każdy wątek obsługuje niezależne urządzenie, identyfikowane unikalnym ID. Wyzwaniem w tym obszarze będzie optymalizacja zużycia procesora (CPU), co może wymagać zastosowania wieloprocesowości (ang. *multiprocessing*) zamiast wielowątkowości, aby ominąć ograniczenia blokady GIL (ang. *Global Interpreter Lock*) w języku Python.

## Bezpieczeństwo i dostęp zdalny

Finalnym etapem rozwoju powinno być utwardzenie bezpieczeństwa aplikacji (ang. *Hardening*) oraz umożliwienie bezpiecznego dostępu spoza sieci lokalnej. Planowane działania obejmują:

- **Bezpieczny tunel:** Zamiast wystawiania portów aplikacji bezpośrednio do Internetu, rekomendowane jest zintegrowanie systemu z rozwiązaniami typu VPN (np. WireGuard) lub tunelami Cloudflare, co pozwoli na bezpieczny dostęp zdalny bez kompromitowania sieci lokalnej.

## Bibliografia

- „An IoT-Based Multimodal Real-Time Home Control System” (b.d.). W: *Intelligent Data Analysis* (). URL: [https://library.acadlore.com/IDA/2023/2/2/IDA\\_02.02\\_04.pdf](https://library.acadlore.com/IDA/2023/2/2/IDA_02.02_04.pdf).
- Antonakakis, Manos i in. (2017). „Understanding the Mirai Botnet”. W: *26th USENIX Security Symposium (USENIX Security 17)*, s. 1093–1110.
- Bella, Giampaolo i in. (2023). „PETIoT: PEneration Testing the Internet of Things”. W: *Internet of Things* 22, s. 100707. DOI: 10.1016/j.iot.2023.100707. URL: <https://doi.org/10.1016/j.iot.2023.100707>.
- British Design Council (2025). *Double Diamond (design process model)*. Version as of October 27, 2025. Accessed: 2026-01-08. URL: <https://www.designcouncil.org.uk/our-resources/the-double-diamond/>.
- Design Council (2021). *Double Diamond by the Design Council*. Accessed: 2026-01-08. URL: [https://commons.wikimedia.org/wiki/File:Double\\_diamond\\_design\\_model\\_2021.jpg](https://commons.wikimedia.org/wiki/File:Double_diamond_design_model_2021.jpg).
- Fette, I. i A. Melnikov (2011). *The WebSocket Protocol*. RFC 6455. URL: <https://www.rfc-editor.org/rfc/rfc6455>.
- Hacefresko (2021). *TP-Link Tapo C200 unauthenticated RCE (CVE-2021-4045)*. Blog post. (Data dostępu: 2026-01-07). URL: <https://www.hacefresko.com/posts/tp-link-tapo-c200-unauthenticated-rce>.
- International Telecommunication Union (2019a). *Recommendation H.264: Advanced video coding for generic audiovisual services*. <https://www.itu.int/rec/T-REC-H.264>.
- (2019b). *Recommendation H.265: High efficiency video coding*. <https://www.itu.int/rec/T-REC-H.265>. Standard ITU-T.
- Moxter, Martin (b.d.). *Camera for monitoring critical infrastructure such as streets, schools, squares, authorities, AI generated*. Adobe Stock. Autor: Martin Moxter/IMAGE-BROKER, plik: 1229288039, dostęp: 2026-01-06. URL: <https://stock.adobe.com/pl/images/camera-for-monitoring-critical-infrastructure-such-as-streets-schools-squares-authorities-ai-generated/1229288039>.
- Neshenko, Nataliia i in. (2019). „Vulnerability Analysis of Low-Cost Consumer IoT Devices”. W: *IEEE Communications Surveys & Tutorials* 21.4, s. 3236–3273.
- Open Web Application Security Project (2018). *OWASP IoT Top 10*. <https://owasp.org/www-project-internet-of-things/>. Dostęp: 2025-10-08.
- Schulzrinne, H., A. Rao i R. Lanphier (1998). *Real Time Streaming Protocol (RTSP)*. RFC 2326. URL: <https://www.ietf.org/rfc/rfc2326.txt>.
- Schulzrinne, H. i in. (2003). *RTP: A Transport Protocol for Real-Time Applications*. RFC 3550.

Tanenbaum, Andrew S i David J Wetherall (2011). "Computer Networks". 5 wyd. Pearson.

;

# **Spis rysunków**

1.1	Monitoring wizyjny z wykorzystaniem kamer IP w infrastrukturze miejskiej. Źródło: Moxter, b.d. . . . .	8
2.1	Model Double Diamond. Źródło: Design Council, 2021. . . . .	46
2.2	Architektura systemu IoT Gateway - przepływ danych między komponentami . . . . .	55
2.3	Diagram UML klas - struktura obiektowa aplikacji . . . . .	56
2.4	Diagram sekwencji - przebiegi procesu nagrywania . . . . .	57
3.1	Fotografia przedstawiająca przebieg badania. . . . .	89
3.2	Wykres opóźnienia end-to-end podczas testu T02. . . . .	92
3.3	Wykres stabilności FPS podczas testu T02. . . . .	93
3.4	Wykres zużycia pamięci RAM podczas testu T03. . . . .	94

## **Spis tabel**

1.1	Główne obszary zastosowań kamer IP w różnych sektorach przemysłu i usług. . . . .	7
1.2	Etapy przetwarzania w potoku ISP. . . . .	17
1.3	Podstawowe komendy protokołu RTSP. . . . .	21
1.4	Kluczowe Specyfikacje Techniczne TP-Link Tapo C200 . . . . .	35
1.5	Analiza Protokołów Komunikacyjnych Tapo C200 pod kątem Integracji Open-Source . . . . .	38
2.1	Podział warstw architektury systemu IoT . . . . .	48
2.2	Podział kompetencji w warstwie multimedialnej . . . . .	63
3.1	Specyfikacja techniczna stacji roboczej (Host). . . . .	86