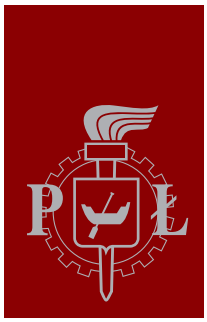

Politechnika Łódzka
Wydział Fizyki Technicznej, Informatyki
i Matematyki Stosowanej

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

Streszczenie

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 (IoT)**. 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 dominacja systemów opartych na **zamkniętym oprogramowaniu (proprietary software)**.

Wybór tematu pracy wynika z konieczności zaadresowania problemu **vendor lock-in** w kontekście popularnych kamer konsumenckich, na przykładzie urządzeń TP-Link Tapo. Zjawisko to, polegające na uzależnieniu pełnej funkcjonalności sprzętu od infrastruktury chmurowej i aplikacji mobilnej producenta, ogranicza **dostepność danych** oraz **możliwości integracji** z otwartymi systemami automatyki i bezpieczeństwa. Problem ten jest szczególnie istotny w kontekście **cyber bezpieczeństwa**, gdzie zamknięte i często nieaudytowane firmware może stanowić potencjalny wektor ataku.

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 **reprodukowalności 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 (*low latency*).

Niniejsza praca ma za zadanie stanowić nie tylko dowód kompetencji inżynierskich, ale także praktyczny wkład w rozwój otwartych technologii w dziedzinie monitoringu IoT.

Cel i zakres pracy

Cel

Celem głównym niniejszej pracy inżynierskiej jest opracowanie oraz implementacja kompletnego, modułowego rozwiązania programistycznego opartego wyłącznie 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 (problem *vendor lock-in*).

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 **reprodukowalności środowiska** na platformach mikserwerowych IoT (np. Raspberry Pi).
- Implementacja funkcjonalności **zapisu nagrań 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 *vendor lock-in* w segmencie kamer IoT, uzasadniając wybór tematu rosnącą potrzebą na otwarte i bezpieczne systemy zarządzania danymi.

Aspekty objęte zakresem pracy

- Projekt trójwarstwowej architektury kontenerowej (Docker) dla warstwy dostępu do sprzętu, logiki biznesowej (Flask/WebSocket's) oraz warstwy prezentacji (Web Client).
- Wykorzystanie biblioteki `PyTapo` do obsługi nieudokumentowanego API komend sterujących kamery (PTZ, tryb nocny, aktywacja usług).
- Praca skupia się na przechwytywaniu jednokierunkowego strumienia wideo i audio. Implementacja pełnej komunikacji zwrotnej w czasie rzeczywistym jest **poza zakresem** projektu.
- Przeprowadzenie **testów wydajnościowych** skupiających się na **zużyciu zasobów (CPU/RAM)** 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:

- Protokół inicjalizacji - **Provisioning** - kamery Tapo w trybie Access Point (AP) jest nieudokumentowany, szyfrowany i opiera się na wymianie kluczy sesyjnych, co wymaga weryfikacji po stronie chmury TP-Link. Odtworzenie tego protokołu jest zadaniem na poziomie projektu badawczego i **wykracza poza ramy niniejszej pracy**. 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 - YOLO), ze względu na wysokie wymagania obliczeniowe i złożoność czasową, **została wykluczona**. Praca koncentruje się na detekcji ruchu opartej na różnicy klatek (`OpenCV`).

1 Wprowadzenie technologiczne Kamer IP

Rozdział ten ma za zadanie ugruntować zrozumienie złożoności systemów kamer IP i precyzyjnie wskazać na luki w otwartych standardach, które musi wypełnić zaprojektowane rozwiązanie. 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 peryferyjne 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.

Obszar zastosowania	**Przykłady wykorzystania kamer IP**
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	Zwiększanie 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.

Zastosowanie monitoringu wizyjnego opartego na kamerach IP jest obecnie wielosektorowe i dynamiczne. Urządzenia te, integrujące funkcje sensora i procesora danych, stały się podstawą **systemów analitycznych** w kluczowych obszarach gospodarki i bezpieczeństwa. W kontekście dalszego rozwoju monitoringu wizyjnego, szczególne znaczenie zyskuje **sztuczna inteligencja (AI)** i **uczenie maszynowe (ML)**. Nowoczesne algorytmy pozwalają na automatyczną detekcję zagrożeń, eliminację fałszywych alarmów oraz identyfikację i śledzenie obiektów w czasie rzeczywistym. Integracja tych zaawansowanych technik z **otwartym oprogramowaniem** — co jest celem niniejszej pracy — otwiera drogę do stworzenia bardziej zaawansowanych, konfigurowalnych i niezależnych narzędzi wspierających bezpieczeństwo oraz analitykę zdarzeń. Dzięki temu, system monitorujący może automatycznie filtrować szum wizualny i koncentrować uwagę na zdarzeniach o wysokim prawdopodobieństwie zagrożenia lub anomalii. Technologie te transformują surowe dane wideo w zorganizowane i użyteczne metadane, co jest fundamentalne dla automatyki i bezpieczeństwa.

1.1.1 Monitoring

Podstawowym i historycznym zastosowaniem kamery IP jest **nadzór wizyjny (monitoring)**. W odróżnieniu od analogowego CCTV, monitoring oparty na protokole internetowym umożliwia przesyłanie strumienia wideo wysokiej rozdzielczości (np. 1080p w Tapo C200) oraz metadanych poprzez standardowe sieci LAN/WLAN. Z technicznego punktu widzenia, monitoring realizowany jest poprzez ciągłe kodowanie wideo (standardy H.264/H.265), strumieniowanie za pomocą protokołów czasu rzeczywistego (**RTSP**) oraz zapis cyfrowy na nośnikach lokalnych (microSD, serwer NVR) lub w chmurze.

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. Dzięki wykorzystaniu AI, kamery stają się kluczowym sensorem w bezdotykowej autoryzacji. Przykłady zastosowań inżynierskich obejmują:

1. **Rozpoznawanie Twarzy:** Zastosowanie algorytmów głębokiego uczenia do identyfikacji i weryfikacji osób uprawnionych, automatycznie odblokowując wejścia.
2. **Rozpoznawanie Tablic Rejestracyjnych:** 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 zabezpieczeń 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:

1. **Kontroli Jakości:** Monitorowanie linii produkcyjnych w celu automatycznego wykrywania defektów, niezgodności montażu lub nieprawidłowej sekwencji działań.
2. **Optymalizacji Przepływu Pracy:** 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 (Edge Analytics)**, 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 inteligentnym, Tapo C200, podobnie jak inne urządzenia IoT, jest zintegrowana za pomocą protokołów API z platformami takimi jak **Google Assistant i Amazon Alexa** (jak wskazano w dokumentacji Tapo). Przykłady zastosowań to:

1. **Automatyzacja Zdarzeniowa:** Detekcja ruchu lub dźwięku (np. wykrywanie płaczu dziecka w Tapo C200) uruchamia inne urządzenia (np. włącza światło, wysyła alert do systemu zarządzania domem).
2. **Zarządzanie Energią:** Wykrycie braku obecności osób w pomieszczeniu może prowadzić do automatycznego obniżenia temperatury lub wyłączenia niepotrzebnych urządzeń, przyczyniając się do zwiększenia efektywności energetycznej.

1.1.5 Analiza Danych

Kamera IP, połączona ze sztuczną inteligencją (AI), 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:

1. **Ekstrakcja Danych Statystycznych:** Dotyczy pomiarów ilościowych, takich jak gęstość obiektów, liczenie przepływu (*flow counting*) oraz generowanie map ciepła (*heatmaps*).

2. **Analiza Behawioralna i Wzorce Trendów:** Identyfikacja nietypowych sekwencji zdarzeń, które mogą sugerować incydent bezpieczeństwa (np. pozostawiony pakunek).

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

1.2 Budowa

Zrozumienie architektury kamery IP jest kluczowe dla identyfikacji ograniczeń narzucanych przez producentów i zaprojektowania skutecznego, otwartego oprogramowania.

Z perspektywy inżynierskiej, kamera IP nie jest monolitycznym urządzeniem, lecz złożonym systemem wbudowanym, składającym się ze ściśle 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

Warstwa sprzętowa stanowi fizyczny fundament kamery, odpowiadając za akwizycję, przetwarzanie i transmisję danych audiowizualnych. W celu dogłębnej analizy, jej kluczowe komponenty zostaną omówione w dedykowanych podrozdziałach.

1.2.1.1 Matryca (Przetwornik obrazu) Matryca, nazywana również przetwornikiem lub sensorem obrazu, jest kluczowym elementem półprzewodnikowym, który inicjuje cały proces wizyjny. Jej fundamentalnym zadaniem jest konwersja energii fotonów (światła) padających na jej powierzchnię na mierzalny sygnał elektryczny. Proces ten, znany jako wewnętrzne zjawisko fotoelektryczne, stanowi podstawę cyfrowego przetwarzania obrazu. We współczesnych kamerach IP, w tym w analizowanym modelu TP-Link Tapo C200, dominującą technologią jest **CMOS (Complementary Metal-Oxide-Semiconductor)**. Przetworniki CMOS wyparły starszą technologię CCD głównie ze względu na niższy koszt produkcji, mniejsze zużycie energii oraz możliwość integracji dodatkowych obwodów logicznych (np. przetworników analogowo-cyfrowych, układów redukcji szumów) bezpośrednio na tej samej płytce krzemowej, co jest zgodne z architekturą System-on-a-Chip (SoC). Z inżynierskiego punktu widzenia, jakość generowanego obrazu jest determinowana przez następujące parametry techniczne matrycy:

- **Rozdzielczość:** Określa liczbę pikseli, z których składa się obraz, np. 1920x1080 (Full HD) w kamerze Tapo C200. Wyższa rozdzielczość pozwala na zarejestrowanie większej liczby szczegółów, ale jednocześnie generuje większy strumień danych, co stanowi wyzwanie dla przepustowości sieci i zasobów obliczeniowych.

- **Rozmiar fizyczny i rozmiar piksela:** Fizyczny rozmiar matrycy (np. 1/2.9 cala) w połączeniu z jej rozdzielczością definiuje rozmiar pojedynczego piksela. Większe piksele są w stanie przechwycić więcej światła, co przekłada się na lepszą jakość obrazu w warunkach słabego oświetlenia i wyższy stosunek sygnału do szumu (SNR).
- **Czułość:** Mierzona w luksach (lux), określa minimalną ilość światła potrzebną do wygenerowania użytecznego obrazu. Jest to parametr krytyczny dla funkcji noktowizyjnych, gdzie matryca musi efektywnie współpracować z oświetlaczem podczerwieni (IR).
- **Zakres dynamiki:** Zdolność matrycy do jednoczesnego rejestrowania bardzo jasnych i bardzo ciemnych obszarów w tej samej scenie. Szeroki zakres dynamiki (WDR) jest kluczowy w scenach o dużym kontraście, np. w pomieszczeniu z oknem w słoneczny dzień.

Zrozumienie tych parametrów jest niezbędne do oceny ograniczeń sprzętowych kamery i świadomego projektowania algorytmów przetwarzania obrazu, takich jak detekcja ruchu, które muszą operować na danych wyjściowych dostarczanych przez matrycę.

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 po włączeniu 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 dogłębną analizę mechanizmów operacyjnych, które definiują funkcjonalność nowoczesnej kamery IP. Urządzenie to, dalekie od bycia prostym peryferium, jest w rzeczywistości zaawansowanym, autonomicznym systemem

wbudowanym, w którym zbiegają się dziedziny inżynierii sprzętowej, oprogramowania firmware oraz złożonych protokołów sieciowych. Celem tej analizy jest dekonstrukcja logiki działania kamery na cztery fundamentalne, wzajemnie powiązane domeny. Rozpoczniemy od zbadania jej architektury komunikacyjnej, czyli stosu protokołów, który umożliwia jej funkcjonowanie jako węzła w sieci. Następnie przeanalizujemy krytyczny proces provisioningu, czyli bezpiecznego włączania urządzenia do infrastruktury sieciowej. W trzeciej części prześledzimy wewnętrzny potok przetwarzania danych, od momentu konwersji zjawisk fizycznych – fotonów światła i fal akustycznych – na surowe dane cyfrowe, aż po ich kompresję do formatu gotowego do transmisji. Na końcu szczegółowo omówimy mechanizmy strumieniowania, które pozwalają na przesyłanie tych danych w czasie rzeczywistym. Zrozumienie synergii między wyspecjalizowanym układem scalonym (SoC), sensorami, oprogramowaniem układowym i protokołami sieciowymi jest kluczowe do pełnego pojęcia, jak kamera IP realizuje swoje zadania – od prostego monitoringu po zaawansowaną analitykę danych.

1.3.1 Architektura Komunikacji Sieciowej: Stos Protokołów

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

1.3.1.1 Warstwa Dostępu do Sieci Warstwa ta definiuje fizyczną transmisję danych. W kamerach Tapo kluczowe są standardy bezprzewodowe oraz mechanizmy zabezpieczeń:

- **IEEE 802.11n/ac:** Standardy Wi-Fi zapewniające przepustowość niezbędną dla strumieni Full HD. Wybór pasma (2.4 GHz vs 5 GHz) determinuje zasięg oraz odporność na zakłócenia.
- **WPA2/WPA3:** Protokoły szyfrowania (AES). WPA3 wprowadza mechanizm *Simultaneous Authentication of Equals*, chroniący przed atakami typu brute-force oraz zapewniający *forward secrecy*.

1.3.1.2 Warstwa Internetowa Odpowiada za adresowanie i routing pakietów w sieciach IP.

- **Adresacja IP:** Kamera uzyskuje tożsamość sieciową dynamicznie (DHCP) lub poprzez stałą konfigurację (statyczny adres IP). W profesjonalnych implementacjach zalecany jest statyczny adres IP, co gwarantuje stabilność połączenia dla zewnętrznych systemów sterujących.

1.3.1.3 Warstwa Transportowa Determinuje sposób przesyłania danych pomiędzy procesami, wykorzystując dwa odmienne podejścia:

- **TCP (Transmission Control Protocol):** Protokół połączeniowy, gwarantujący dostarczenie i kolejność pakietów. Wykorzystywany do zadań krytycznych: obsługi interfejsu konfiguracyjnego oraz przesyłania poleceń sterujących PTZ (*Pan-Tilt-Zoom*) poprzez HTTP API.
- **UDP (User Datagram Protocol):** Protokół bezpołączeniowy o niskim narzucie, optymalny dla transmisji czasu rzeczywistego. Stanowi fundament dla protokołu RTP, umożliwiając strumieniowanie wideo przy minimalnych opóźnieniach kosztem dopuszczalnej utraty pojedynczych klatek.

1.3.1.4 Warstwa Aplikacji Najwyższa warstwa realizująca konkretne funkcje systemowe:

- **HTTP API:** Interfejs programistyczny umożliwiający zewnętrznym aplikacjom (np. napisanym w Pythonie) sterowanie kamerą i odczyt jej stanu.
- **NTP (Network Time Protocol):** Kluczowy dla synchronizacji czasu, co jest niezbędne do prawidłowego generowania znaczników czasu (*timestamps*) w strumieniu wideo oraz logowania zdarzeń o charakterze kryminalistycznym.

Tabela 1.2: Porównanie protokołów transportowych w systemie kamery IP.

Cecha	TCP	UDP (RTP)
Zastosowanie	Sterowanie (API), konfiguracja	Strumieniowanie mediów
Zaleta	Niezawodność i integralność danych	Minimalne opóźnienia (*latency*)
Narzut	Większy (retransmisje, potwierdzenia)	Minimalny (brak kontroli błędów)

1.3.2 Provisioning: Inicjalizacja i Uwierzytelnianie Urządzenia

Zanim nowa kamera IP stanie się funkcjonalnym i zaufanym elementem sieci, musi przejść przez krytyczny proces zwany provisioningiem. Nie jest to jedynie techniczna konfiguracja, ale fundamentalny proces ustanawiania cyfrowej tożsamości urządzenia i zakotwiczania zaufania, który przekształca anonimowy sprzęt prosto z pudełka w zweryfikowany i bezpieczny węzeł sieciowy.

1.3.2.1 Definicja i Cel Provisioningu

Provisioning (w polskim kontekście często nazywany inicjalizacją, udostępnianiem lub aprowizacją) to kompleksowy proces bezpiecznego wprowadzania nowego urządzenia do środowiska sieciowego. Obejmuje on cały cykl życia, od pierwszego uruchomienia, poprzez konfigurację, uwierzytelnienie, aż po zarządzanie operacyjne i ewentualne wycofanie z użytku. Głównym celem provisioningu jest zapewnienie, że tylko autoryzowane, bezpieczne i prawidłowo skonfigurowane urządzenia uzyskują dostęp do sieci i jej zasobów. Jest to pierwsza i najważniejsza linia obrony w ekosystemie Internetu Rzeczy (IoT), gdzie potencjalnie miliony urządzeń mogą stanowić wektor ataku, jeśli nie zostaną prawidłowo zweryfikowane.

Proces ten opiera się na fundamentalnej zasadzie bezpieczeństwa znanej jako Zero Trust. Sieć nie ufa żadnemu urządzeniu domyślnie, nawet jeśli znajduje się ono fizycznie w jej zasięgu. Każde urządzenie musi najpierw udowodnić swoją tożsamość i uzyskać autoryzację, zanim zostanie dopuszczone do komunikacji. W przypadku kamer IP, których strumień wideo jest daną wrażliwą, solidny proces provisioningu jest absolutnie kluczowy.

1.3.2.2 Przykładowy Proces Provisioningu dla Kamery Wi-Fi

Proces provisioningu dla typowej konsumenckiej kamery Wi-Fi, takiej jak modele TP-Link Tapo, jest zaprojektowany tak, aby był jak najprostszy dla użytkownika końcowego, jednocześnie realizując niezbędne kroki bezpieczeństwa. Zazwyczaj odbywa się on za pośrednictwem dedykowanej aplikacji mobilnej producenta i można go podzielić na trzy główne etapy.

1. Rejestracja (Enrollment) Pierwszym krokiem jest zarejestrowanie fizycznego urządzenia w systemie zarządzania producenta.

- **Tryb Access Point (AP):** Po pierwszym podłączeniu do zasilania, kamera nie próbuje łączyć się z żadną istniejącą siecią. Zamiast tego, uruchamia własną, tymczasową sieć Wi-Fi o niewielkim zasięgu, działając w trybie punktu dostępowego (Access Point). Ta sieć jest zazwyczaj otwarta lub zabezpieczona prostym, domyślnym hasłem.
- **Połączenie z Aplikacją:** Użytkownik, postępując zgodnie z instrukcjami w aplikacji mobilnej, łączy swój smartfon z tą tymczasową siecią Wi-Fi emitowaną przez kamerę. W tym momencie smartfon i kamera znajdują się w tej samej, izolowanej sieci, co pozwala aplikacji na bezpośrednie "odkrycie"kamery i nawiązanie z nią bezpiecznej komunikacji.
- **Identyfikacja Urządzenia:** Aplikacja odczytuje unikalny identyfikator sprzętowy kamery (np. adres MAC lub numer seryjny) i rejestruje go na koncie użytkownika

w chmurze producenta. Ten krok formalnie przypisuje to konkretne urządzenie do tego konkretnego użytkownika.

2. Konfiguracja (Configuration) Po nawiązaniu bezpośredniego połączenia, aplikacja mobilna przekazuje kamerze niezbędne dane konfiguracyjne, aby mogła ona funkcjonować w docelowej sieci.

- **Przekazanie Poświadczeń Sieciowych:** Najważniejszym elementem tego etapu jest bezpieczne przekazanie kamerze nazwy (SSID) i hasła do domowej sieci Wi-Fi użytkownika. Aplikacja szyfruje te dane i wysyła je bezpośrednio do kamery.
- **Ustawienia Dodatkowe:** W tym kroku mogą być również przekazywane inne ustawienia, takie jak nazwa kamery (np. "Salon"), strefa czasowa, a także może zostać zainicjowana automatyczna aktualizacja oprogramowania firmware do najnowszej wersji.

3. Uwierzytelnianie (Authentication) Jest to kulminacyjny i najważniejszy z punktu widzenia bezpieczeństwa etap provisioningu.

- **Restart i Połączenie z Siecią Docelową:** Po otrzymaniu konfiguracji, kamera kończy działanie w trybie AP i restartuje się. Następnie próbuje połączyć się z domową siecią Wi-Fi, używając otrzymanych poświadczeń.
- **Uwierzytelnianie w Chmurze:** Po pomyślnym połączeniu z siecią lokalną i uzyskaniu dostępu do internetu, kamera nawiązuje połączenie z serwerami chmurowymi producenta. W tym momencie następuje kluczowy proces uwierzytelniania. Kamera przedstawia serwerowi swój unikalny, wbudowany fabrycznie certyfikat cyfrowy (np. w standardzie X.509). Ten certyfikat działa jak niezaprzeczalny, kryptograficzny dowód tożsamości.
- **Weryfikacja i Udzielenie Dostępu:** Serwer chmurowy weryfikuje ten certyfikat, sprawdzając, czy pochodzi on z zaufanego źródła (czyli od samego producenta) i czy odpowiada urządzeniu, które zostało wcześniej zarejestrowane na koncie użytkownika. Jak zauważono w dokumentacji źródłowej, cały ten proces jest szyfrowany i wymaga weryfikacji po stronie chmury TP-Link, co świadczy o jego złożoności i krytycznym znaczeniu. Dopiero po pomyślnej weryfikacji certyfikatu, serwer uznaje kamerę za w pełni uwierzytelnioną i zaufaną. Od tego momentu kamera uzyskuje pełen dostęp do usług chmurowych (np. zdalnego podglądu, powiadomień push) i staje się w pełni funkcjonalnym urządzeniem.

Ten wieloetapowy proces, choć dla użytkownika sprowadza się do kilku kliknięć w aplikacji, jest w rzeczywistości starannie zaprojektowaną sekwencją operacji kryptograficznych i sieciowych. Wbudowany fabrycznie certyfikat pełni rolę "cyfrowego aktu urodzenia"kamery, jednoznacznie i niepodrabialnie poświadczając jej pochodzenie. Pomyślne

uwierzytelnienie w chmurze jest momentem, w którym ta tożsamość zostaje oficjalnie potwierdzona, a urządzenie otrzymuje "pozwolenie na pracę"w sieci. Jest to fundamentalny mechanizm, który chroni zarówno użytkownika, jak i całą infrastrukturę IoT przed wprowadzeniem do niej fałszywych lub skompromitowanych urządzeń.

1.3.3 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 (pipeline), realizowanym w czasie rzeczywistym przez wyspecjalizowane komponenty sprzętowe wewnątrz układu System-on-a-Chip (SoC). Architektura SoC jest tu kluczowa; zamiast obciążać uniwersalny procesor (CPU) 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 AV.

1.3.3.1 Ścieżka Przetwarzania Obrazu (Image Pipeline) Droga, jaką przebywa informacja wizualna od obiektywu do interfejsu sieciowego, jest najbardziej złożonym procesem wewnątrz kamery.

1.3.3.2 Ścieżka Przetwarzania Obrazu (Image Pipeline) 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) Gonzalez i Woods, 2018.

- **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 natężenie światła (luminancję). 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 Gonzalez i Woods, 2018. W rezultacie, na wyjściu z matrycy otrzymujemy surowy, "mozaikowy"obraz o pikselach tylko o jednym kolorze.

- **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 (proces A/D) jeszcze na poziomie samego sensora lub w jego bezpośrednim sąsiedztwie.

Tabela 1.3: Etapy przetwarzania w potoku ISP.

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

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:

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 (AVC) lub H.265 (HEVC) 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 (pomiędzy kolejnymi klatkami). Analizują obraz w poszukiwaniu podobnych bloków i zamiast przesyłać pełną informację o każdym z nich, przesyłają tylko informację o różnicach i wektorach ruchu Wiegand i in., 2003.
- **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.

Na wyjściu z kodera otrzymujemy skompresowany elementarny strumień wideo (elementary stream), którego przepływność jest zredukowana stukrotnie lub więcej (np. do kilku Mb/s), co umożliwia jego efektywną transmisję przez sieć Sullivan i in., 2012.

1.3.3.3 Ścieżka Przetwarzania Dźwięku (Audio Pipeline)

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 jednobitowy 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 (zmniejszenia częstotliwości próbkowania). W rezultacie jednobitowy strumień PDM o wysokiej częstotliwości jest konwertowany na standardowy, wielobitowy (np. 16-bitowy) 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, i oferuje lepszą jakość przy tej samej przepływności w porównaniu do starszego formatu MP3.

1.3.3.4 Synchronizacja i Muksowanie Ostatnim krokiem wewnątrz 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 (timestamps) 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.

1.3.4 Strumieniowanie: 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 strumieniowanie (streaming), opiera się na zestawie wyspecjalizowanych protokołów warstwy aplikacji, które zarządzają sesją i transportują dane w sposób zoptymalizowany pod kątem czasu rzeczywistego.

1.3.4.1 Separacja Sterowania i Danych: Rola RTSP i RTP Fundamentalną zasadą architektoniczną w strumieniowaniu na żywo jest rozdzielenie płaszczyzny sterowania (control plane) od płaszczyzny danych (data plane). 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 optymalizację każdego z tych zadań z osobna: sterowanie wymaga niezawodności, a przesyłanie danych – szybkości i niskich opóźnień.

- **RTSP (Real-Time Streaming Protocol):** Pełni rolę "sieciowego pilota zdalnego sterowania". 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. "zacznij 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 multimedialnychSchulzrinne, Rao i Lanphier, 1998.
- **RTP (Real-time Transport Protocol):** Jest to protokół odpowiedzialny za transport danych. Jego zadaniem jest opakowanie skompresowanych danych wideo (H.264) i audio (AAC) 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.Schulzrinne i in., 2003

1.3.4.2 Nawiązywanie Sesji Strumieniowej: Uścisk Dłoni RTSP Zanim na ekranie klienta pojawi się pierwszy obraz, musi on przeprowadzić z kamerą negocjacje za pomocą protokołu RTSP. Ten proces, często nazywany "uściskiem dłoni"(handshake),

przebiega w kilku krokach i jest niezbędny do ustalenia parametrów transmisji.Schulzrinne, Rao i Lanphier, 1998

Tabela 1.4: Podstawowe komendy protokołu RTSPSchulzrinne, Rao i Lanphier, 1998.

Komenda	**Nadawca**	**Cel**
DESCRIBE	Klient	Żądanie od serwera (kamery) opisu dostępnych strumieni multimedialnych. Odpowiedź zawiera dane w formacie **SDP**, informujące np. o istnieniu strumienia wideo H.264 i audio AAC.
SETUP	Klient	Konfiguracja transportu dla każdego strumienia z osobna. Klient informuje serwer, na których portach **UDP** będzie nasłuchiwał na pakiety *RTP* (dane) i *RTCP* (dane kontrolne).
PLAY	Klient	Polecenie dla serwera, aby rozpoczął transmisję pakietów *RTP* na wcześniej uzgodnione porty.
PAUSE	Klient	Wstrzymanie transmisji strumienia bez zrywania sesji. Sesję można wznowić komendą **PLAY**.
TEARDOWN	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`). Kamera odpowiada, wysyłając opis w formacie SDP (Session Description Protocol), który informuje klienta, że dostępne są dwa strumienie: jeden wideo (zakodowany w H.264) i jeden audio (zakodowany w AAC).

2. **SETUP:** Klient, chcąc odbierać oba strumienie, wysyła dwa osobne żądania SE-

TUP – jedno dla strumienia wideo 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.

1.3.4.3 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.Schulzrinne i in., 2003

1.3.4.4 Monitorowanie Jakości Strumienia (RTCP) Równolegle z wysyłaniem danych przez RTP, działa protokół RTCP (Real-time Transport Control Protocol). Jest to protokół towarzyszący RTP, który służy do przesyłania informacji kontrolnych i statystyk dotyczących sesji. W przeciwieństwie do jednokierunkowego przepływu danych RTP (z kamery do klienta), komunikacja RTCP jest dwukierunkowa. Klient okresowo wysyła do kamery raporty (Receiver Reports) zawierające informacje o jakości odbioru, takie jak liczba utraconych pakietów, miara jittera czy czas podróży w obie strony (round-trip time). Te informacje zwrotne są niezwykle cenne. Zaawansowana kamera lub serwer strumieniujący może na ich podstawie dynamicznie dostosowywać parametry transmisji, na przykład obniżając bitrate (jakość) strumienia wideo w odpowiedzi na wykryte

przeciążenie sieci, aby zapewnić ciągłość transmisji kosztem jakości obrazu. RTCP jest więc mechanizmem zapewniającym adaptacyjność i odporność strumieniowania na zmienne warunki sieciowe.

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, inteligentnej 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 i wartości inżynierskiej 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 znacząco 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.

Od strony technicznej, mechanizm PTZ opiera się na precyzyjnych, miniaturowych silnikach krokowych, które wykonują polecenia otrzymywane z oprogramowania sterującego. Sterowanie odbywa się zdalnie za pomocą dedykowanych aplikacji, oprogramowania VMS (Video Management System) lub fizycznych kontrolerów z joystickiem.

Komunikacja ta jest realizowana za pomocą różnych protokołów, od starszych standardów szeregowych jak RS-485, po nowoczesne protokoły sieciowe, takie jak ONVIF (Open Network Video Interface Forum) czy własnościowe API producenta oparte na HTTP.

1.4.2 Wykrywanie obiektów i zdarzeń - AI

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, SSD) 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 wałęsania 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 Noktowizja i termowizja

Zdolność do "widzenia"w ciemności jest kluczową funkcją kamer bezpieczeństwa. Realizowana jest ona głównie za pomocą dwóch odrębnych technologii: noktowizji w podczerwieni oraz termowizji.

1.4.4.1 Noktowizja w podczerwieni (IR Night Vision) Jest to najpopularniejsza technologia stosowana w kamerach konsumenckich i profesjonalnych. Jej działanie opiera się na oświetleniu sceny za pomocą diod LED emitujących światło w paśmie bliskiej podczerwieni (IR), które jest niewidoczne dla ludzkiego oka, ale doskonale "widziane"przez matrycę kamery. Kluczowym elementem jest tutaj mechaniczny filtr odcinający podczerwień (IR Cut Filter).

- **W dzień:** Filtr jest umieszczony między obiektywem a matrycą i blokuje światło podczerwone, które zniekształciłoby kolory. Dzięki temu obraz ma naturalne, wierne barwy.

- **W nocy:** Gdy czujnik światła wykryje niski poziom oświetlenia, filtr jest mechanicznie odsuwany. Jednocześnie aktywowane są diody IR, a kamera przełącza się w tryb monochromatyczny (czarno-biały), który jest znacznie bardziej czuły na światło podczerwone. Pozwala to na uzyskanie wyraźnego obrazu nawet w całkowitej ciemności.

1.4.4.2 Termowizja (Thermal Imaging) Jest to zupełnie inna, bardziej zaawansowana technologia. Kamera termowizyjna nie potrzebuje żadnego źródła światła. Zamiast tego, jej specjalny sensor (mikrobolometr) wykrywa promieniowanie cieplne (daleką podczerwień) emitowane przez wszystkie obiekty, których temperatura jest wyższa od zera absolutnego. Obraz jest tworzony na podstawie różnic temperatur – cieplejsze obiekty, takie jak ludzie czy zwierzęta, są wyraźnie widoczne na tle chłodniejszego otoczenia. Główne zalety termowizji to:

- Działanie w absolutnej ciemności i trudnych warunkach atmosferycznych (mgła, dym, deszcz).
- Wysoka skuteczność w wykrywaniu intruzów na dużych odległościach i w ukryciu (np. w zaroślach).
- Mniejsza liczba fałszywych alarmów, ponieważ nie reaguje na cienie, odbicia światła czy ruch obiektów nieożywionych.

Jednak kamery termowizyjne są znacznie droższe i zazwyczaj oferują niższą rozdzielczość, co uniemożliwia identyfikację szczegółów, takich jak rysy twarzy.

1.4.5 Dwukierunkowe audio

Funkcja dwukierunkowego audio przekształca kamerę z pasywnego urządzenia nasłuchowego w interaktywny interkom. 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 werbalnego ostrzeżenia potencjalnego intruza, co często jest skutecznym środkiem prewencyjnym.

1.4.6 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.

1.4.6.1 Zapis lokalny na karcie microSD Wiele kamer, zwłaszcza z segmentu konsumenckiego, jest wyposażonych w gniazdo na kartę pamięci microSD. Umożliwia to zapis nagrań 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 nagrań w przypadku kradzieży lub fizycznego uszkodzenia samej kamery.

1.4.6.2 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.

1.4.6.3 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.

1.4.7 Integracja z Inteligentnymi Systemami

Siła kamer IP leży w ich zdolności do bycia częścią większego, zintegrowanego ekosystemu. Otwartość protokołów sieciowych umożliwia komunikację z szeroką gamą innych urządzeń i platform programistycznych. Kluczowe technologie umożliwiające integrację to:

- **RTSP (Real-Time Streaming Protocol):** Standardowy protokół, który pozwala zewnętrznym aplikacjom i urządzeniom (np. odtwarzaczom wideo, systemom NVR) na dostęp do strumienia wideo z kamery. Jest to fundamentalny element interoperacyjności.
- **ONVIF (Open Network Video Interface Forum):** Globalny standard mający na celu ujednolicenie komunikacji między urządzeniami do nadzoru wideo różnych producentów. Kamera zgodna z ONVIF może być łatwo zintegrowana z systemem VMS lub NVR innej firmy, co daje użytkownikowi swobodę wyboru komponentów systemu.
- **API (Application Programming Interface):** Wielu producentów udostępnia własne API, które pozwala na programistyczną kontrolę zaawansowanych funkcji kamery, niedostępnych w standardzie ONVIF. To właśnie takie API jest wykorzystywane w niniejszej pracy do sterowania kamerą Tapo.

Dzięki tym mechanizmom, kamera IP może stać się inteligentnym czujnikiem w systemie automatyki domowej. Przykładowo, wykrycie ruchu przez kamerę w ogrodzie po zmroku może automatycznie włączyć oświetlenie zewnętrzne, a obraz z kamery przy drzwiach może być wyświetlany na inteligentnym ekranie po naciśnięciu dzwonka.

1.4.8 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 narzucane 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: rozdzielczości, liczby klatek na sekundę (FPS), zastosowanego kodeka kompresji oraz złożoności obserwowanej sceny.

- **Rozdzielczość (Resolution):** Wyższa rozdzielczość oznacza większą liczbę pikseli w każdej klatce, co przekłada się na bardziej szczegółowy obraz, ale jednocześnie wykładniczo zwiększa ilość danych do przesłania. Strumień wideo w roz-

dzielczości 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 (FPS):** 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 (Compression):** 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 (Scene Complexity):** 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.

Te wymagania stają się szczególnie problematyczne w kontekście zapisu w chmurze. Większość konsumenckich planów internetowych ma charakter asymetryczny, oferując wysoką prędkość pobierania (download), ale znacznie niższą prędkość wysyłania (upload). Ponieważ kamera wysyła strumień wideo do chmury, kluczowa jest właśnie przepustowość wysyłania. Pojedyncza kamera 4K może z łatwością wysycić całe dostępne pasmo wysyłania typowego łącza domowego, uniemożliwiając lub znacznie spowalniając działanie innych usług internetowych, takich jak wideokonferencje czy wysyłanie dużych plików.

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 w środowiskach bezprzewodowych (Wi-Fi) są nie wyjątkiem, a regułą.

- **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 (Interference):** Sygnały z sąsiednich sieci Wi-Fi, urządzeń Bluetooth, kuchenek 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.
    - **Kongestia sieciowa (Network Congestion):** 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 Zmienność 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 multiplikatywny. 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 konsu-

menckich – łatwość instalacji dzięki łączności bezprzewodowej – stoi w bezpośredniej sprzeczności z ich technicznym wymaganiem posiadania stabilnej, wysokoprzepustowej i niskoprzetłoczonej sieci. Użytkownikowi sprzedawany jest produkt o wysokiej rozdzielczości, który w docelowym, typowym środowisku domowej sieci Wi-Fi, rzadko ma szansę osiągnąć swoją nominalną jakość działania.

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 cybernetycznych. Ich umiejscowienie 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 Powszechne 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 skanery, 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 (port forwarding), 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.

- **Wykorzystanie jako proxy do działalności przestępczej:** Nowszym i bardziej podstępnym zagrożeniem jest wykorzystywanie przejętych kamer jako serwerów proxy do anonimizacji działalności przestępczej. Badania naukowe dowodzą, że skompromitowane urządzenia IoT, w tym w dużej mierze kamery, są masowo wykorzystywane w infrastrukturze przestępczej do przeprowadzania ataków na instytucje finansowe, takich jak credential stuffing (automatyczne testowanie skradzionych loginów i haseł), kradzież kryptowalut czy oszustwa z użyciem kart kredytowych Mi i in., 2019. Właściciel kamery jest najczęściej nieświadomy, że jego domowe urządzenie zabezpieczające stało się węzłem w globalnej sieci przestępczej.

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. Połączenie zamkniętego, nieaudytowanego kodu z nieregularnym wsparciem ze strony producenta tworzy trwałą i niebezpieczną powierzchnię ataku.

- **Zamknięty i nieprzejrzysty 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), zaprzestaje 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 wię-

cej, 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 udokumentowanych 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 chmurowy. 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 nagrań 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 priorytetyzuje sprzedaż nowego sprzętu nad wspieraniem istniejących produktów. Tworzy to zjawisko "planowanego starzenia się bezpieczeństwa", 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 załatania 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 ekosystemów. Działania te, motywowane biznesowo, w sposób fundamentalny ograniczają prawa użytkownika, interoperacyjność i długoterminowe bezpieczeństwo, co stanowi główną motywację dla projektu badawczego opisanego w niniejszej pracy.

1.5.3.1 Zjawisko "Vendor Lock-in"w Ekosystemach IoT

Producenci celowo projektują swoje produkty w taki sposób, aby stworzyć wysokie koszty zmiany dostawcy,

uzależniając klienta od swojego ekosystemu na cały cykl życia produktu Opara-Martins, Sahandi i Tian, 2016.

- **Definicja i mechanizm:** Zjawisko "vendor lock-in"(uzależnienie od dostawcy) ma miejsce, gdy koszt i wysiłek związany ze zmianą produktu na konkurencyjny są tak znaczące, że klient jest w praktyce "uwięziony"u pierwotnego dostawcy Opara-Martins, Sahandi i Tian, 2016. W przypadku kamer IP jest to realizowane poprzez ścisłe powiązanie sprzętu (kamery) z dedykowanym oprogramowaniem (aplikacją mobilną) i usługami backendowymi (platformą chmurową) producenta. Funkcje kluczowe, takie jak pierwsza konfiguracja, zdalny podgląd, powiadomienia o ruchu czy zapis w chmurze, są dostępne wyłącznie za pośrednictwem tego zamkniętego ekosystemu.
- **Konsekwencje dla użytkownika:** Użytkownik nie ma możliwości integracji i zarządzania kamerami różnych marek w jednej, wspólnej aplikacji. Jeśli zdecyduje się na zmianę platformy (np. z powodu niezadowolenia z usług lub polityki cenowej), często jest zmuszony do wymiany całego posiadanego sprzętu, nawet jeśli jest on w pełni sprawny technicznie **alfuqaha2015iot**.

1.5.3.2 Konsekwencje Zamkniętych Protokółów i API

Głównym narzędziem technicznym służącym do egzekwowania strategii "vendor lock-in"jest stosowanie zamkniętych, nieudokumentowanych interfejsów programistycznych (API) zamiast otwartych, ustandaryzowanych protokołów.

- **Blokowanie interoperacyjności:** Chociaż istnieją globalne standardy, takie jak ONVIF, stworzone w celu zapewnienia współpracy urządzeń różnych producentów, wielu dostawców sprzętu konsumenckiego celowo ich nie implementuje lub oferuje jedynie częściowe, zawodne wsparcie.
- **Własnościowa kontrola:** Zamiast tego, do sterowania zaawansowanymi funkcjami, takimi jak ruch PTZ, zmiana ustawień czy dostęp do funkcji opartych na AI, wykorzystywane są prywatne, nieudokumentowane API.
- **Konieczność inżynierii wstecznej:** Aby zintegrować taką kamerę z systemem open-source (np. Home Assistant) lub z autorskim rozwiązaniem, takim jak opracowane w ramach niniejszej pracy, programiści są zmuszeni do prowadzenia złożonego i czasochłonnego procesu inżynierii wstecznej w celu rozszyfrowania działania zamkniętych protokołów. Takie rozwiązanie jest z natury niestabilne, gdyż każda aktualizacja oprogramowania firmware przez producenta może zmienić API i zniszczyć działającą integrację.

1.6 Wnioski - 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** (*pipeline*), który w czasie rzeczywistym dokona translacji strumieni do formatów kompatybilnych z technologiami takimi jak **WebSockets** czy **HTML5 Canvas**.

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 zaawansowana detekcja zdarzeń, 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 wstecznej** 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 (*AI*),
- integracja z systemami *Smart Home*,

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

2 Analiza Kamery TP-Link TAPO C200

2.1 Charakterystyka Ogólna i Pozycja Rynkowa

Kamera TP-Link Tapo C200 jest pozycjonowana na rynku jako flagowy przykład konsumenckiego urządzenia **Internetu Rzeczy (IoT)** w kategorii **„Smart Home"**. Jej podstawowym celem rynkowim 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 rozdzielczość 1080p Full HD (1920×1080 pikseli) przy płynnej prędkości 30 klatek na sekundę. Stanowi to standard rynkowy dla nowoczesnych systemów monitoringu, pozwalający na wyraźną identyfikację szczegółów.
- **Mechanizm Pan/Tilt (PTZ):** Urządzenie jest wyposażone w zmotoryzowaną głowicę, umożliwiającą zdalny obrót w poziomie (Pan) w zakresie 360° oraz pochylenie w pionie (Tilt). Ta funkcja eliminuje martwe strefy i pozwala na monitorowanie całego pomieszczenia za pomocą jednego urządzenia.
- **Tryb nocny (Noktowizja):** Zintegrowane diody LED podczerwieni (IR) 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)**.

Należy jednak podkreślić, że zamierzony przez producenta model operacyjny (**Intended Operational Model**) jest fundamentalnie oparty na koncepcji **„zamkniętego ogrodu" (walled garden)**. 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.

2.2 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.

Sercem każdej kamery IP jest jej przetwornik obrazu. Tapo C200 wykorzystuje sensor $1/2.8''$ **Progressive Scan CMOS**. Jest to kluczowa informacja, ponieważ rozmiar sensora i typ technologii CMOS determinują bazową jakość obrazu, czułość na światło (kluczową dla noktowizji) oraz zakres dynamiczny. Jest to fundament, na którym opiera się cały potok przetwarzania wideo.

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 $IEEE802.11$b/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 demozaikowanie i redukcja szumów, oraz – co najważniejsze – sprzętowy koder wideo H.264.

Wybór takiej architektury SoC jest kluczową decyzją inżynieryjną 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.

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

Tabela 2.1: 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 (zasięg do 40 stóp / 12 m)
Rozdzielczość	1080P HD (1920×1080 px)
Szybkość Klatek	30 fps
Kompresja Wideo	H.264
System Audio	Wbudowany mikrofon i głośnik
Standard Wi-Fi	$IEEE802.11$b/g/n, 2.4 GHz
Zapis Lokalny	Gniazdo microSD (do 512 GB)

2.3 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 (**closed-source**) 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. Jak wykazano w sekcji 2.6, ten monolityczny i nieaudytowalny charakter firmware'u jest sam w sobie znaczącym wektorem ataku.

Standardowy Stos Sieciowy i „Iluzja Otwartości”

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 niemal wyłącznie dla serwerów chmurowych TP-Link.

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

RTSP (Real-Time Streaming Protocol): Specyfikacja techniczna potwierdza wsparcie dla RTSP. Jest to absolutnie kluczowy, otwarty i ustandaryzowany protokół, który pozwala na dostęp do surowego, skompresowanego strumienia wideo (H.264) i audio. Dostępność strumienia RTSP jest fundamentalnym umożliwiaczem (enabler) dla całego projektu niniejszej pracy. To właśnie ten protokół pozwala narzędziom takim jak FFmpeg i OpenCV na przechwycenie obrazu i jego dalszą analizę w sposób całkowicie niezależny od ekosystemu producenta.

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. Jak potwierdzają badania oraz liczne raporty społeczności open-source, implementacja ONVIF w Tapo C200 jest celowo okrojona. Ogranicza się ona w najlepszym razie do minimalnego zestawu funkcji (np. Profile S, co oznacza jedynie możliwość udostępniania strumienia wideo, co i tak jest już realizowane przez RTSP). Co najważniejsze, implementacja ta nie udostępnia kluczowej funkcjonalności sterowania PTZ.

Rzeczywisty Mechanizm Sterowania: Własnościowe API Skoro ONVIF nie pozwala na sterowanie kamerą, powstaje pytanie, w jaki sposób realizuje to oficjalna aplikacja Tapo. Odpowiedź leży w istnieniu nieudokumentowanego, własnościowego (**proprietary**) protokołu sterowania.

Badania społecznościowe wykazały, że aplikacja mobilna Tapo komunikuje się z kamerą w sieci lokalnej za pomocą niestandardowego, opartego na HTTP (lub HTTPS)

API. Wysyła ona zaszyfrowane lub zakodowane żądania w celu wykonania operacji takich jak ruch Pan/Tilt, włączenie trybu nocnego, czy zmiana ustawień detekcji.

Ten zamknięty protokół jest technicznym narzędziem egzekwowania **„vendor lock-in”**. Ponieważ jest nieudokumentowany i może ulec zmianie przy każdej aktualizacji firmware'u, uniemożliwia on standardowym, otwartym platformom (jak Home Assistant, ZoneMinder czy openHAB) natywną kontrolę nad urządzeniem.

To właśnie ta bariera zrodziła potrzebę inżynierii wstecznej (**reverse-engineering**) po stronie społeczności. Biblioteka PyTapo, która jest jednym z kluczowych narzędzi wykorzystywanych w niniejszej pracy, jest bezpośrednim rezultatem tego procesu. PyTapo implementuje logikę tego nieudokumentowanego protokołu, hermetyzując jego złożoność i pozwalając na programistyczne sterowanie kamerą z poziomu Pythona. Zależność niniejszej pracy od PyTapo jest sama w sobie dowodem na istnienie i celowość bariery w postaci zamkniętego API.

Poniższa tabela podsumowuje krytyczną analizę protokołów komunikacyjnych kamery.

Tabela 2.2: 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 (*FFmpeg/OpenCV*).
ONVIF	Interoperacyjność (*Stream* + Sterowanie)	Otwarty Standard	Pozornie Tak	Znikoma. Implementacja jest okrojona i nie udostępnia sterowania *PTZ*. Nazywana „openwashingiem”.
Proprietary API	Pełne sterowanie urządzeniem (*PTZ*, ustawienia)	Zamknięty / Własnościowy	Brak informacji	Kluczowa (Pośrednio). Wymaga **inżynierii wstecznej**. Projekt wykorzystuje *PyTapo* do obsługi tego API.
Protokół Chmurowy (*HTTPS*)	Zdalny dostęp, alerty, *provisioning*	Zamknięty / Własnościowy	Brak informacji	Brak. Jest to mechanizm, który projekt ma na celu ominąć, aby **uniezależnić się od producenta**.

2.4 Analiza Możliwości Funkcjonalnych

Sekcja ta dokonuje ponownej oceny funkcji reklamowanych w sekcji 2.1, tym razem przez pryzmat inżynierski, oceniając 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. Dostęp do nich jest strzeżony przez własnościowe API.

W konsekwencji, z perspektywy dewelopera open-source, kamera Tapo C200 bez dodatkowej inżynierii wstecznej 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: Problem „Czarnej Skrzynki"

Najbardziej złożona sytuacja dotyczy wbudowanych funkcji „AI Detection", takich jak wykrywanie osób i płaczu dziecka. Stanowią one istotę „Wyzwania Open Source" (tytuł sekcji 2.3 w).

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 wideo (przez RTSP) i przeprowadzić własną,

serwerową analizę detekcji ruchu (np. za pomocą OpenCV). Jest to kluczowe uzasadnienie dla jednego z głównych celów szczegółowych pracy – implementacji własnego algorytmu detekcji.

2.5 Ograniczenia i Zjawisko „Vendor Lock-in”

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

Krytyka tego modelu biznesowego wskazuje, że kamera jest traktowana jako niskomarżowy **„koń trojański”**. Rzeczywistym celem nie jest jednorazowa sprzedaż sprzętu, ale **„uwięzienie”** użytkownika w zamkniętym ekosystemie Tapo, co otwiera drogę do generowania przychodów cyklicznych, np. poprzez sprzedaż subskrypcji na przechowywanie nagrań w chmurze (Tapo Care).

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 rozwiązaniach, 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.

2.6 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.

Ryzyka dla Prywatności

Model operacyjny oparty na chmurze zmusza użytkownika do fundamentalnego kompromisu w zakresie prywatności. Wymaga on przesyłania wrażliwych danych – strumieni audio i wideo z wnętrza prywatnego domu – na serwery firmy trzeciej. Taka architektura generuje trzy główne 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 strumieni.
- **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 (KRYTYCZNA) w skali CVSS.

- **Problem:** Luka typu „unauthenticated Remote Code Execution” (nieuwierzytelnione zdalne wykonanie kodu).
- **Wektor:** Luka znajduje się w binarnym pliku uhttpd – tym samym wbudowanym serwerze WWW, który jest używany do obsługi... własnościowego API sterującego.
- **Wpływ:** Serwer uhttpd działa z uprawnieniami użytkownika root (najwyższymi możliwymi). Oznacza to, że nieuwierzytelniony atakujący w tej samej sieci (np. gość korzystający z Wi-Fi) mógł zdalnie przejąć całkowitą kontrolę nad kamerą. Mógł ją wyłączyć, podsłuchiwać, podglądać, a także – co być może najgroźniejsze – wykorzystać ją jako „przyczółek” (beachhead) do ataku na inne urządzenia w sieci lokalnej użytkownika (np. komputer lub dysk NAS).
- **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 Ariefianto / Biondi et al., w ramach którego opracowano metodykę PETIoT, wykorzystało Tapo C200 jako studium przypadku i zidentyfikowało trzy nieznane wcześniej (zero-day) luki: Denial of Service (DoS), podsłuchiwanie strumienia wideo (video eavesdropping) oraz nowy typ ataku nazwany „Motion Oracle”.
   - Inna praca dyplomowa (KTH) przeprowadzająca testy penetracyjne C200, zidentyfikowała podatności na ataki typu brute force, RCE, Man-in-the-Middle (MITM) oraz replay attack, wskazując na fundamentalne problemy z szyfrowaniem firmware’u i protokołami komunikacyjnymi.
   - 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.

Wniosek Końcowy: Związek „Vendor Lock-in” z Lukami w Zabezpieczeniach

Niniejsza analiza wykazuje istnienie bezpośredniego związku przyczynowego między modelem biznesowym „vendor lock-in” a katastrofalnymi lukami bezpieczeństwa.

Logika jest następująca:

1. Aby zrealizować strategię „vendor lock-in”, TP-Link musiał zrezygnować z otwartego standardu ONVIF do sterowania.

2. Wymusiło to stworzenie własnościowego, zamkniętego API.

3. Aby to API było dostępne, kamera musi uruchamiać niestandardowy, wbudowany serwer WWW (uhttpd).

4. Aby ten serwer mógł kontrolować sprzęt (silniki PTZ, diody IR), musiał otrzymać najwyższe uprawnienia systemowe (root).

5. W ten sposób stworzono idealny wektor ataku: skomplikowaną, nieaudytowalną, autorską usługę sieciową działającą z maksymalnymi uprawnieniami.

6. Dokładnie w tym miejscu – w serwerze uhttpd – odkryto krytyczną lukę RCE (CVE-2021-4045).

Wniosek: To nie przypadek. To decyzja biznesowa o zamknięciu ekosystemu bezpośrednio doprowadziła do stworzenia architektury oprogramowania, która była fundamentalnie niebezpieczna.

Ostateczne Uzasadnienie dla Projektu

Powyższa analiza bezpieczeństwa i prywatności stanowi ostateczne i najsilniejsze uzasadnienie dla celu niniejszej pracy. Projektowane rozwiązanie open-source nie jest jedynie ćwiczeniem z inżynierii wstecznej 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/wideo nigdy nie opuszczają sieci lokalnej.
- Neutralizuje ryzyko botnetu: Kamera nie może komunikować się z serwerami C&C (Command and Control).
- Omija podatny na ataki serwer: Użytkownik komunikuje się z bezpiecznym, audytowalnym serwerem Python (rozwiązaniem z pracy), zamiast z dziurawym, działającym jako root uhttpd.

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.

3 Metodologia i implementacja rozwiązania

Poprzednie rozdziały dokonały teoretycznej dekonstrukcji technologii kamer IP (Rozdział 1) oraz przeprowadziły szczegółową analizę studium przypadku — kamery TP-Link Tapo C200 (Rozdział 2). 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 szczegóły implementacji poszczególnych komponentów. Celem jest budowa autorskiego, otwartego rozwiązania programistycznego, które uwalnia pełen potencjał kamery i realizuje cele postawione w niniejszej pracy.

3.1 Metodyka Projektowa

3.1.1 Double Diamond

Model Double Diamond (Podwójny Diament) jest ustrukturyzowaną metodyką procesową, pierwotnie sformalizowaną przez British Design Council w 2005 roku. 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. Zespół projektowy 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. Zespół filtruje i analizuje informacje, szukając wzorców i kluczowych

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

Przestrzeń Rozwiązania - Diament Drugi

3. **Faza Rozwijania -** Mając jasno zdefiniowany problem, zespół ponownie przechodzi w tryb research, aby wygenerować jak najszerszy 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 ostatenicznie wybrac optymalne rozwiązania gotowego do wdrożenia.

Zastosowanie modelu w niniejszej pracy

1. **Odkrywanie:** Tę fazę reprezentuje research przeprowadzony w Rozdziałach 1 i 2. Zbadano ogólne działanie kamer IP, a następnie przeanalizowano specyfikę Tapo C200, identyfikując jej otwarte porty (RTSP) oraz zamknięte, własnościowe API do sterowania PTZ.

2. **Definiowanie:** Zebrane informacje skumulowano do konkretnego problemu: **vendor lock-in** uniemożliwia lokalną kontrolę. Celem pracy stało się więc stworzenie lokalnego systemu dającego pełną kontrolę.

3. **Rozwój:** W tej fazie nastąpił brainstorming nad architekturą rozwiązania. Rozważano różne technologie i narzędzia (np. gotowe platformy vs. własna aplikacja).

4. **Dostarczanie:** Wybrano konkretne, optymalne rozwiązanie: aplikacja webowa oparta na Pythonie, Flasku i WebSockets, wykorzystująca bibliotekę PyTapo (do sterowania) oraz OpenCV i PyAV, całość hermetyzowana w Dockerze. Implementacja tego rozwiązania stanowi dalszą część niniejszego rozdziału.

3.2 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 (Klient), Logikę Aplikacji (Middleware) oraz Warstwę Danych(Źródło). Taka separacja zapewnia, że złożoność własnościowych protokołów kamery zostaje całkowicie abstrahowana i ukryta przed interfejsem użytkownika końcowego.

Ponieważ nowoczesne przeglądarki internetowe nie są w stanie natywnie obsługiwać surowych strumieni wideo RTSP ani komunikować się bezpośrednio za pomocą

protokołu ONVIF, warstwa pośrednia działa jako dwukierunkowy translator protokołów. Pobiera ona synchroniczne strumienie sprzętowe i przekształca je w asynchroniczne zdarzenia. Pozwala to na uzyskanie responsywnego doświadczenia użytkownika bez konieczności ujawniania danych uwierzytelniających sprzętu czy jego adresu IP w sieci publicznej.

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

Kolejnym wzorcem architektonicznym jest **Architektura Sterowana Zdarzeniami** (*Event-Driven Architecture*), która umożliwia komunikację w czasie rzeczywistym między klientem a serwerem. Akcje użytkownika oraz zmiany stanu systemu są propagowane natychmiastowo, zamiast polegać na cyklicznym odpytywaniu (*polling*).

Ze względu na ciągły charakter strumieniowania wideo, architektura systemu w znacznym stopniu polega na **wielowątkowości** (*threaded concurrency*). Aplikacja utrzymuje współdzielony stan w pamięci operacyjnej (*shared in-memory state*), co pozwala na odseparowanie szybkich 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.

Rysunek 3.1: Schemat architektury rozwiązania (Klient - Middleware - Sprzęt)

3.2.1 Architektura Wielowarstwowa

Współczesna inżynieria systemów **Internetu Rzeczy**, a w szczególności projektowanie bram sieciowych (*IoT Gateways*) obsługujących strumieniowanie multimediów 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.

W kontekście systemów IoT, model ten ewoluuje w kierunku struktur typu **Edge-Fog-Cloud**, gdzie brama (*Gateway*) pełni rolę kluczowego węzła pośredniczącego. 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 3.1) przedstawia szczegółowy podział odpowiedzialności w poszczególnych warstwach systemu.

Tabela 3.1: Podział warstw architektury systemu IoT

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

3.2.1.1 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, realizując paradygmat tzw. **„cienkiego klienta”** (*Thin Client*). Jej odpowiedzialność ograniczona jest wyłącznie do dwóch funkcji:

- **Wizualizacja danych:** Prezentowanie wyników przetwarzania dostarczanych przez warstwę biznesową (obraz z kamery, statusy czujników, lista nagrań).

- **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 technologiach 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 (**Canvas**) 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 Kliencka (JavaScript):** Skrypty uruchamiane w przeglądarce, odpowiedzialne za **dynamiczną aktualizację treści** bez przeładowywania strony. Ich rola ogranicza się do nasłuchiwania na kanały komunikacyjne (**WebSockets**) i natychmiastowego odświeżania elementów *DOM* 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.

3.2.1.2 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 (IoT) warstwa ta jest często definiowana jako ***Middleware*** (oprogramowanie pośredniczące), którego fundamentalnym zadaniem jest ukrycie złożoności urządzeń końcowych i udostępnienie ujednoliconych 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.

Model Współbieżności i 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 (*Data Plane*):** 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 (*Control Plane*):** Obsługuje zdarzenia inicjowane przez użytkownika, takie jak sterowanie mechaniką kamery (PTZ) czy przełączanie stanu nagrywania. Płaszczyzna ta priorytetyzuje krótki czas reakcji (niską latencję).
- **Płaszczyzna Komunikacji:** Realizuje warstwę transportową, wykorzystując protokoły WebSocket do komunikacji dwukierunkowej (Full-Duplex) oraz HTTP do serwowania zasobów statycznych. Łączy ona warstwę prezentacji z logiką serwera.

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 (RTSP, kodeki H.264, dźwięk próbkowany w 44.1kHz Float32), 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 (*Downscaling*) oraz rekompresji do formatu JPEG. Operacja ta ma na celu dostosowanie przepustowości strumienia do możliwości sieciowych klienta, zapewniając responsywność interfejsu (*Low Latency*) nawet przy słabszym łączu internetowym.
- **Normalizacja Audio:** Warstwa ta rozwiązuje problem niekompatybilności formatów dźwięku. Surowe dane z kamery (często w formacie stereo lub o zmiennej częstotliwości) są konwertowane do ustandaryzowanego formatu PCM (Mono, 16kHz). Zapobiega to powstawaniu artefaktów dźwiękowych (zniekształcenia, szumy statyczne) po stronie klienta i gwarantuje poprawną interpretację sygnału przez Web Audio API.

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

- **Sterowanie PTZ (*Pan-Tilt-Zoom*):** Moduł sterowania działa jako „tłumacz" intencji użytkownika. Odbiera on wysokopoziomowe polecenia (np. „przesuń w górę"), waliduje je pod kątem bezpieczeństwa (np. sprawdzając, czy nie trwa już inny ruch), a następnie zleca wykonanie sekwencji instrukcji do sterownika silników.
- **Zarządzanie Nagrywaniem:** Zaimplementowano maszynę stanów (*State Machine*), która kontroluje proces rejestracji. Logika ta zarządza buforowaniem danych w pamięci RAM, synchronizacją ścieżek audio/wideo (*Lip-Sync*) oraz finalną kompilacją 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" (*Gatekeeper*) dla systemu sprzętowego, implementując wzorzec **API Gateway**. 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.

Reasumując, Warstwa Logiki odpowiada za **Orkiestrację Danych**. Zapewnia ona, że szybkie i synchroniczne strumienie danych napływające z warstwy sprzętowej są odpowiednio buforowane, przetwarzane oraz bezpiecznie dostarczane do asynchronicznego klienta przeglądarkowego przy pomocy odpowiednich wątków.

3.2.1.3 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 (wideo, audio, telemetria) 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 „pod spodem" wykonuje całą komunikacyjną „brudną robotę".

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

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

Centralizację obsługi błędów: HAL tłumaczy specyficzne wyjątki sieciowe (np. `ConnectionRefusedEr` czy kody błędów z serwera `uhttpd` kamery) na zrozumiałe wyjątki domenowe (np. `CameraOfflineException`), 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. Jest to kluczowe w kontekście znanych podatności kamer Tapo, takich jak `CVE-2021-4045` (luka RCE w serwerze `uhttpd`), która wymusza traktowanie urządzenia jako potencjalnie niebezpiecznego i izolowanie interakcji z nim.

3.2.2 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.

3.2.2.1 Zastosowanie w projekcie W zrealizowanym systemie wzorzec ten stanowi fundament działania klas `VideoStreamer` oraz `AudioStreamer`, które operują w nieskończonych pętlach wątków tła. Ponieważ dane z kamery (protokół RTSP) 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 miksowanie kanałów dźwiękowych.
- **Potok Rejestracji:** Buforowanie ramek w pamięci RAM i ich finalna kompozycja do pliku MP4 (realizowana przez bibliotekę `MoviePy`).

3.2.2.2 Przykład implementacji: Potok przetwarzania wideo Najbardziej reprezentatywnym przykładem wykorzystania tego wzorca w projekcie jest pętla przetwarzania obrazu zaimplementowana w klasie `VideoStreamer`. Proces ten można przedstawić jako sekwencję pięciu filtrów:

1. **Źródło (*Source*):** Metoda `camera.read_frame()` dokonuje akwizycji surowej klatki obrazu bezpośrednio ze sterownika sprzętowego.

2. **Filtr Optymalizacyjny:** Surowy obraz, często o wysokiej rozdzielczości natywnej, jest poddawany operacji skalowania (`cv2.resize`). Zmniejszenie rozdzielczości na tym etapie jest krytyczne dla wydajności kolejnych kroków analizy i transmisji.

3. **Filtr Analityczny (*Motion Detection*):** Przeskalowana klatka trafia do modułu `MotionDetector`. Jest ona porównywana z modelem tła (średnią kroczącą z poprzednich klatek). Wynikiem tego filtra nie jest modyfikacja obrazu, lecz wygenerowanie metadanych (flaga `is_motion`), które sterują logiką powiadomień.

4. **Filtr Kodujący:** Obraz będący macierzą pikseli (format BGR) jest kompresowany do formatu JPEG (`cv2.imencode`). Jest to niezbędny krok transformacji danych do formatu zrozumiałego dla przeglądarek internetowych.

5. **Ujście (*Sink*):** Zakodowany obraz, wraz z metadanymi o detekcji ruchu, jest przekazywany do warstwy transportowej (`socketio.emit`), która dystrybuuje go do wszystkich podłączonych klientów.

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.

3.2.3 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* – EDA). 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 sekwencyjny kod proceduralny.

3.2.3.1 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 na nadejście sygnału. Wzorzec ten składa się z trzech głównych elementów:

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

Kanał Zdarzeń (*Event Channel*): Medium transportowe, które przekazuje zdarzenie od producenta do konsumenta. W projekcie rolę tę pełni biblioteka `Flask-SocketIO` działająca na protokole **WebSocket**.

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

3.2.3.2 Zastosowanie w projekcie W zrealizowanym systemie bramy IoT, architektura sterowana zdarzeniami została wykorzystana jako główny mechanizm komunikacji dwukierunkowej (*Full-Duplex*) między Warstwą Prezentacji (przeglądarką) a Warstwą Logiki (serwerem Python). Zastosowanie tego wzorca było niezbędne do osiągnięcia niskiej latencji (opóźnienia) wymaganej 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 (np. zmiana koloru diody nagrywania) ze stanem procesu rejestracji wideo na serwerze.

Dzięki luźnemu powiązaniu komponentów (*loose coupling*), serwer może obsłużyć setki takich zdarzeń na sekundę, zapewniając płynne sterowanie „oko-ręka”, niemożliwe do osiągnięcia w architekturze synchronicznej.

3.3 Diagramy

Diagram komponentow

Diagram klas

Diagram komunikacji

3.4 Zastosowane narzędzia i technologie

Niniejszy rozdział stanowi dogłębną analizę techniczną **stosu technologicznego** (*technology stack*) 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 biblioteki specyficzne dla domeny przetwarzania sygnałów.

3.4.1 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. Python, dzięki swojemu dojrzałemu ekosystemowi, pełni w projektowanym systemie rolę **warstwy orkiestracji** (ang. *glue code*).

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 (*I/O bound*), 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 wytwórczy oprogramowania. W kontekście integracji z urządzeniami IoT, pozwala to na elastyczne dostosowywanie protokołów komunikacyjnych i logiki sterowania bez konieczności długotrwałej rekompilacji całego projektu.

3.4.2 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 (*feedback loop*) 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 inżynieryjny projektu, analogiczny do `Cargo.lock` w Rust.

Zaawansowana integracja z Dockerem

W projekcie wykorzystano specyficzne techniki optymalizacji współpracy `uv` z systemem plików Docker (*OverlayFS*). Zastosowanie mechanizmu montowania *cache'u* (*BuildKit cache mounts*) pozwala na **persystencję pobranych artefaktów** pomiędzy kolejnymi budowaniami kontenera. Dodatkowo, strategia **Bytecode Compilation** wspierana natywnie przez `uv` skraca czas startu aplikacji (*cold start*), co jest istotne w przypadku restartu usługi na urządzeniu monitoringu.

3.4.3 Ekosystem Konteneryzacji

Wdrożenie oprogramowania na **urządzeniach brzegowych** (*Edge Devices*) wiąże się z wyzwaniami heterogeniczności sprzętowej i konfliktów bibliotecznych. 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 podwyższonego ryzyka cybernetycznego (patrz: analiza podatności *CVE* w Rozdziale 2). Uruchomienie autorskiego serwera sterującego bezpośrednio na systemie operacyjnym hosta (*bare-metal*) niosłoby ryzyko, że ewentualne przejęcie kontroli nad aplikacją dałoby atakującemu dostęp do całego systemu. Docker zapewnia silną **izolację procesów** wykorzystując mechanizmy jądra **Linux** (*cgroups*, *namespaces*).

W projekcie zastosowano dodatkowo praktykę „non-root user" wewnątrz kontenera oraz **minimalizację uprawnień** (*capabilities drop*), co drastycznie redukuje powierzchnię ataku i chroni system hosta.

Multi-stage Builds i Optymalizacja Rozmiaru

Urządzenia klasy *embedded* często dysponują ograniczoną przestrzenią dyskową. Aby pogodzić wymagania posiadania ciężkich narzędzi kompilacji (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 kopiowane 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.

3.4.4 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 ze względu na narzut sieciowy (*overhead*).

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 „cienkiego 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*) 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 (*Low-Latency Streaming*):** Klatki wideo są przesyłane jako **binarne ładunki** (*binary payloads*) przez otwarty socket, co pozwala na redukcję opóźnień transmisji (*latency*).
- **Sterowanie Czasu Rzeczywistego (*Real-Time Control*):** Komendy sterujące **PTZ** (Pan/Tilt/Zoom) są przesyłane jako lekkie obiekty **JSON**. Czas reakcji kamery jest zminimalizowany ($< 100ms$).

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** (*Document Object Model*) jest szybsza niż mechanizmy wirtualnego DOM.
- **Redukcję długu technologicznego:** Kod klienta nie wymaga procesu kompilacji (*transpilacji/bundlingu*).
- **Intuicyjną obsługę:** Natywne `EventListeners` służą do przechwytywania zdarzeń klawiatury (sterowanie kamerą za pomocą strzałek).

3.4.5 Biblioteki Przetwarzania Multimediów

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 podyktowana była jej pozycją jako **standardu przemysłowego** oraz kompleksowością oferowanych rozwiązań. W ramach opracowanego oprogramowania, OpenCV realizuje pełen cykl życia danych wizyjnych:

- **Akwizycja Obrazu (*Video Acquisition*):** Wykorzystanie interfejsu `cv2.VideoCapture` pozwala na **stabilne nawiązanie połączenia** ze strumieniem **RTSP** kamery.

- **Przetwarzanie Macierzowe i Analityka:** Po pobraniu klatki, OpenCV wykonuje na niej operacje „inteligentne" (*Smart Features*). Zaimplementowany **algorytm detekcji ruchu** (oparty na odejmowaniu tła i filtracji Gaussa) oraz nanoszenie metadanych (*OSD*) są realizowane bezpośrednio na obiektach tej biblioteki.

- **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** (*binding* dla **FFmpeg**). Jej rola w projekcie jest ściśle zdefiniowana i ograniczona do: **Równoległego nawiązania połączenia**, **Ekstrakcji, dekodowania i transkodowania strumienia audio** (z formatów 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 3.2: Podział kompetencji w warstwie multimedialnej

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

3.4.6 Kontrola Kamery i Inżynieria Wsteczna

Realizacja nadrzędnego 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 sterujących ładunków danych (*payloads*) w formacie **JSON**. Komunikacja ta jest zabezpieczona 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 liczby losowe (*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.

3.4.7 Narzędzie do Kompozycji i Zapisu Danych

Ostatnim ogniwem w łańcuchu przetwarzania danych multimedialnych jest moduł odpowiedzialny za **trwały zapis** (*persystencję*) 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** (*muxing*), czyli scalenia danych wizyjnych (**macierzy NumPy** z OpenCV) i danych fonicznych (**próbek audio** z PyAV) zgromadzonych w buforach pamięci. Wynikiem jest **enkapsulacja** do standardowego **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 (*wrapper*) na oprogramowanie **FFmpeg**. Zastosowanie jej eliminuje złożoność bezpośredniego wywoływania komend FFmpeg i zapewnia automatyczną **synchronizację A/V** (*lip-sync*), zarządzając osią czasu i dopasowując długość ścieżki audio do sekwencji wideo. To kluczowe w przypadku **de-**

tekcji ruchu, gdzie nagrania mają zmienną długość. Proces kodowania odbywa się w sposób **wsadowy** (*batch processing*) w momencie zakończenia nagrania.

Rozszerzalność (Extensibility)

Biblioteka ta oferuje bogaty zestaw funkcji do **nieliniowego montażu wideo** (*NLE*) z poziomu kodu, co ułatwia przyszły rozwój oprogramowania i implementację dodatkowych funkcjonalności, takich jak:

- **Dynamiczne dodawanie znaków wodnych** (*Watermarking*).
- **Łączenie (konkatenacja)** wielu klipów zdarzeń w jeden raport wideo.
- **Nakładanie napisów końcowych** z parametrami zdarzenia (data, typ wykrytego obiektu).

3.5 Proces implementacji rozwiazania

Niniejszy podrozdział stanowi techniczną dokumentację procesu transformacji koncepcji architektonicznej, zdefiniowanej w sekcji 3.4, w w pełni funkcjonalny **artefakt programistyczny**. Celem poniższego opisu jest przedstawienie ewolucyjnego **cyklu wytwarzania oprogramowania** (*SDLC*), który doprowadził do powstania 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** i warstwę komunikacyjną (*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 **FFmpeg**,
- opracowanie **algorytmów detekcji ruchu** działających na brzegu sieci (*edge processing*),
- **konteneryzacja aplikacji** z wykorzystaniem środowiska **Docker**, zapewniająca jej przenośność i powtarzalność wdrożenia.

Opisany poniżej proces stanowi syntezę doboru odpowiednich narzędzi (język Python, framework Flask, protokół WebSocket) oraz metodologii inżynierskiej, prowadzącą do uzyskania gotowego narzędzia nadzoru wizyjnego, niezależnego od chmury producenta.

3.5.1 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 dla 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*). Domyślna konfiguracja *firmware'u* jest nastawiona wyłącznie na komunikację z chmurą producenta. W stanie „po wyjęciu z pudełka" (*OOBE – Out-of-box experience*) kamera charakteryzuje się następującymi **ograniczeniami**:

- Brak otwartego dostępu do strumieniowania protokołem **RTSP** (*Real Time Streaming Protocol*).
- Zablokowane porty odpowiedzialne za standard **ONVIF**.
- Brak zdefiniowanego lokalnego użytkownika administracyjnego, co uniemożliwia autoryzację zewnętrznych skryptów sterujących.

Procedura konfiguracji i odblokowania dostępu

W celu przystosowania urządzenia do współpracy z autorskim rozwiązaniem *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 Kamery" (*Camera Account*):** 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" /

ONVIF. Operacja ta skutkowała otwarciem portu **554** (dla strumienia wideo RTSP) oraz portu **2020** (dla protokołu ONVIF).

> **Uwaga inżynierska:** Aby zapewnić stabilność połączenia dla serwera Python, na routerze sieci lokalnej wykonano **rezerwację adresu IP** (*Static DHCP Lease*) dla adresu MAC kamery.

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.

3.5.2 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.

Inicjalizacja 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).
- `pytapo` – biblioteka kliencka do komunikacji z kamerą.
- `moviepy` oraz `numpy` – operacje na plikach wideo 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.
- `config.json`: Zewnętrzny plik konfiguracyjny (adres IP kamery, poświadczenia).
- `Dockerfile` oraz `uv.lock/pyproject.toml`.

- **Moduł aplikacji (`app/`):**

  - `init.py`: Fabryka aplikacji **Flask**, inicjująca instancję serwera.
  - `settings.py` oraz `shared.py`: Moduły do ładowania konfiguracji i **współdzielenia obiektów** (np. instancji kamery) 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

3.5.3 Implementacja serwera HTTP

Centralnym punktem logicznym systemu, spajającym warstwę prezentacji z logiką backendową, jest serwer HTTP zrealizowany w oparciu o mikroframework **Flask**. Wybór tego rozwiązania podyktowany był koniecznością zachowania **niskiego narzutu pamięciowego** (*low footprint*) przy jednoczesnej elastyczności w obsłudze protokołów asynchronicznych.

Rozwiązanie problemu „Double Execution” i limity sprzętowe

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

Standardowy tryb deweloperski frameworka Flask wykorzystuje mechanizm *reloader*, który monitoruje zmiany w kodzie i powoduje uruchomienie **dwóch procesów systemowych**: procesu monitorującego oraz procesu roboczego. W kontekście aplikacji IoT skutkowało to zjawiskiem „podwójnego uruchomienia” (*double execution*) wątków strumieniujących. Efektem było natychmiastowe **wyczerpanie puli dostępnych połączeń** kamery (*Resource Exhaustion*) i odrzucanie prób autoryzacji.

W celu wyeliminowania tego błędu, w głównym punkcie wejścia aplikacji (`run.py`) wymuszono konfigurację serwera z flagą:

```
if __name__ == '__main__':
    socketio.run(app, debug=True, use_reloader=False)
```

Listing 3.1: Wymuszenie jednokrotnego uruchomienia serwera Flask

Ustawienie parametru `use_reloader=False` zapewniło **deterministyczne, jednokrotne uruchomienie** wątków `VideoStreamer` i `AudioStreamer`, gwarantując stabilność połączenia z kamerą przy zachowaniu pełnej kontroli nad zasobami sieciowymi.

3.5.4 Implementacja Warstwy Dostępu do Sprzętu (HAL)

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 administracyjnego **API HTTP/ONVIF**. Dzięki takiemu podejściu, pozostałe moduły systemu (np. detektor ruchu czy interfejs webowy) 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()` (Listing 3.1) 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`.

Listing 3.1. Implementacja inicjalizacji połączenia hybrydowego (RTSP + HTTP) w klasie `TapoCamera`.

```
class TapoCamera:
    def __init__(self, ip, user, password, cloudPassword):
        self.ip = ip
        self.user = user
        self.password = password
        self.cloudPassword = cloudPassword
        self.cap = None
        self.admin = None

    def connect(self):
```

```
        # 1. Setup Video (RTSP)
        url = f"rtsp://{self.user}:{self.password}@{self.ip}/
            stream1"
        self.cap = cv2.VideoCapture(url)
        video_success = self.cap.isOpened()

        # 2. Setup Controls (Pytapo)
        try:
            self.admin = Tapo(self.ip, self.user, self.
                cloudPassword)
            # Weryfikacja polaczenia poprzez proste zapytanie
            self.admin.getBasicInfo()
            control_success = True
        except Exception as e:
            print(f"Control Error: {e}")
            control_success = False

        if video_success and control_success:
            print(f"Fully Connected to {self.ip}")
            return True
        else:
            return False
```

Źródło: Opracowanie własne.

3.5.5 Abstrakcja sterowania mechaniką (PTZ)

Istotnym wyzwaniem inżynierskim była implementacja sterowania silnikami *Pan-Tilt-Zoom* (**PTZ**) w sposób bezpieczny dla mechaniki urządzenia. Bezpośrednie wywoływanie metod biblioteki `PyTapo` obarczone jest ryzykiem przekroczenia **fizycznego zakresu obrotu** głowicy.

W celu rozwiązania tego problemu zaimplementowano metodę `move()`, która pełni rolę tłumacza (*wrappera*). Konwertuje ona semantyczne polecenia kierunkowe (np. „up”, „left”) na wektory przesunięcia silników. Metoda ta zawiera również **mechanizm obsługi wyjątków** (*Exception Handling*), który przechwytuje informacje o osiągnięciu limitu obrotu („limit_reached”) i zwraca je w ustrukturyzowanej formie do klienta API, zamiast przerywać działanie aplikacji.

Listing 3.2. Implementacja metody sterującej z obsługą błędów granicznych.

```
    def move(self, direction, step=5):
```

```
        """Wykonuje ruch kamera. Zwraca slownik ze statusem operacji."""
        if not self.admin:
            return {"success": False, "error": "not_connected"}

        step = int(step)

        try:
            if direction == "up":
                self.admin.moveMotor(0, step)
            elif direction == "down":
                self.admin.moveMotor(0, -step)
            elif direction == "left":
                self.admin.moveMotor(-step, 0)
            elif direction == "right":
                self.admin.moveMotor(step, 0)

            return {"success": True}

        except Exception as e:
            error_msg = str(e).lower()
            # Sprawdzenie slow kluczowych oznaczajacych koniec zakresu ruchu
            if "range" in error_msg or "limit" in error_msg:
                print(f" Limit Reached: {direction}")
                return {"success": False, "error": "limit_reached"}

            try:
                self.admin = Tapo(self.ip, self.user, self.password)
            except:
                pass
            return {"success": False, "error": "unknown"}
```

Źródło: Opracowanie własne.

3.5.6 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

Klasa `VideoStreamer` inicjowana jest z referencjami do obiektu kamery (*warstwa HAL*) oraz instancji `socketio` (*warstwa komunikacyjna*). Uruchomienie procesu strumieniowania następuje poprzez metodę `start_streaming()`, która powołuje nowy wątek (`threading.Thr` w trybie **demona** (`daemon=True`). Taka konfiguracja gwarantuje, że proces strumieniowania zostanie automatycznie zakończony wraz z zamknięciem głównego procesu aplikacji, zapobiegając powstawaniu „wątków zombie".

Listing 3.5. Inicjalizacja i uruchomienie wątku strumieniującego wideo.

```
class VideoStreamer:
    def __init__(self, camera, socketio):
        self.camera = camera
        self.socketio = socketio
        self.is_running = False
        self.thread = None

        # Inicjalizacja detektora ruchu
        self.detector = MotionDetector(min_area=1000)
        self.motion_active = False

    def start_streaming(self):
        if not self.is_running:
            self.is_running = True
            self.thread = threading.Thread(target=self.
               _capture_loop)
            self.thread.daemon = True
            self.thread.start()
```

Listing 3.2: Inicjalizacja i uruchomienie wątku VideoStreamer

Źródło: Opracowanie własne.

Potok przetwarzania obrazu (Pipeline)

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

1. **Akwizycja i Dystrybucja Wewnętrzna:** Pętla pobiera surową klatkę (*RAW frame*) z kamery. Obraz jest natychmiast przekazywany do **Modułu Nagrywania** (`shared.recorder.add_`) oraz do **Modułu Detekcji** (`self.detector.detect`). Wynik analizy steruje emisją zdarzenia `motion_status` do klienta.

2. **Optymalizacja Transmisji (Skalowanie):** 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** (*latency*).

3. **Kompresja i Serializacja:** Przeskalowana klatka jest poddawana **kompresji stratnej** do formatu **JPEG** (`cv2.imencode`). Uzyskany bufor binarny jest następnie kodowany 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 *framerate* do około **25 klatek na sekundę**, stabilizując obciążenie serwera.

Listing 3.6. Implementacja pętli przetwarzania obrazu (`_capture_loop`).

```
1   def _capture_loop(self):
2       while self.is_running:
3           # 1. Pobranie surowej klatki
4           raw_frame = self.camera.read_frame()
5
6           if raw_frame is not None:
7               # 2. Przekazanie do nagrywarki (jesli aktywna
                    )
8               if shared.recorder and shared.recorder.
                    is_recording:
9                   shared.recorder.add_frame(raw_frame)
10
11              # --- Analiza Detekcji Ruchu na surowej
                    klatce ---
```

```
12                  is_motion, _ = self.detector.detect(raw_frame
                        )
13
14                  # Logika zmiany stanu (emisja tylko przy
                        zmianie)
15                  if is_motion != self.motion_active:
16                      self.motion_active = is_motion
17                      self.socketio.emit('motion_status', {'
                            motion': self.motion_active})
18
19                  # 3. Skalowanie dla Web (Optymalizacja pasma)
20                  web_frame = cv2.resize(raw_frame, (1000, 562)
                        )
21
22                  # 4. Kompresja i Emisja
23                  success, buffer = cv2.imencode('.jpg',
                        web_frame)
24                  if success:
25                      # Kodowanie do Base64 dla transportu
                            tekstowego
26                      b64_frame = base64.b64encode(buffer).
                            decode('utf-8')
27                      self.socketio.emit('video_frame', {'frame
                            ': b64_frame})
28
29              # Taktowanie petli (~25 FPS)
30              self.socketio.sleep(0.04)
```

Listing 3.3: Pętla przetwarzania i transmisji wideo

Źródło: Opracowanie własne.

3.5.7 Realizacja Strumieniowania Audio (Inżynieria Dźwięku)

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.

Ograniczenia biblioteczne i problem „Demon Voice”

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

- **Stan źródłowy:** Kamera Tapo przesyłała dźwięk w formacie wysokiej jakości (np. 44.1 kHz lub 48 kHz).
- **Stan odbiorczy:** Prosty odtwarzacz PCM w przeglądarce klienta oczekiwał domyślnie strumienia o parametrach **16 kHz** (16000 próbek na sekundę).

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 (*Resampling* & *Mixing*)

W celu wyeliminowania opisanych zniekształceń, w pętli przetwarzania audio zaimplementowano proces **transkodowania w czasie rzeczywistym**. Wykorzystano klasę `av.AudioResampler` (Listing 3.7), wymuszając konwersję każdego pakietu do ściśle zdefiniowanego formatu docelowego: **16000 Hz, Mono, s16** (*Signed 16-bit Integer*).

Listing 3.7. Konfiguracja resamplera wymuszająca format zgodny z Web Audio API.

```
    def _stream_loop(self):
        # ... (inicjalizacja polaczenia)

        # Konfiguracja Resamplera:
        # - format='s16': Wymuszenie 16-bitowych liczb
           calkowitych (standard PCM)
        # - layout='mono': Redukcja do jednego kanalu
        # - rate=16000: Downsampling do 16kHz w celu
           oszczednosci pasma
        resampler = av.AudioResampler(format='s16', layout='
           mono', rate=16000)

        for packet in container.demux(stream):
            for frame in packet.decode():
                # Probkowanie klatki do formatu docelowego
                output_frames = resampler.resample(frame)
```

Listing 3.4: Konfiguracja AudioResampler

Źródło: Opracowanie własne.

Kolejnym wyzwaniem była obsługa wielokanałowości. Zamiast prostego odrzucenia jednego kanału, zastosowano **cyfrowe miksowanie kanałów** (*Downmixing*) z wykorzystaniem biblioteki **NumPy**.

Algorytm (Listing 3.8) oblicza **średnią arytmetyczną** z obu kanałów (`np.mean(array, axis=0)`), tworząc zbalansowany sygnał monofoniczny. Dodatkowo, kluczowym krokiem była **jawna konwersja typów danych** z formatu *Float32* (FFmpeg) do wymaganego formatu *Int16* (`np.int16`), co zapobiegało generowaniu **silnego szumu statycznego** (*static noise*) po stronie klienta.

Listing 3.8. Algorytm miksowania kanałów i konwersji typów danych.

```
                for out_frame in output_frames:
                    array = out_frame.to_ndarray()

                    # 1. Miksowanie kanalow (Stereo -> Mono)
                    # Sprawdzenie czy macierz posiada wiecej
                       niz jeden wymiar/kanal
                    if array.ndim == 2:
                        if array.shape[0] > 1:
                            # Obliczenie sredniej z kanalow (
                               Planar Audio)
                            array = np.mean(array, axis=0)
                        elif array.shape[1] > 1:
                            array = array.reshape(-1)

                    # 2. Konwersja formatu (Float32 -> Int16)
                    # Zapobieganie szumom kwantyzacji i
                       bledom interpretacji
                    array = array.astype(np.int16)

                    # Emisja gotowego bufora bajtow
                    self.socketio.emit('audio_chunk', array.
                       tobytes())
```

Listing 3.5: Miksowanie i konwersja do Int16

Źródło: Opracowanie własne.

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

przeglądarek internetowych.

3.5.8 Implementacja Warstwy Komunikacyjnej (Middleware)

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 jednoczesnego przesyłania strumieni multimedialnych (*downlink*) oraz odbierania poleceń sterujących (*uplink*), kluczowym zadaniem było dobranie odpowiedniego protokołu transportowego.

Protokół WebSockets i Flask-SocketIO

W klasycznych systemach CCTV często stosuje się technikę *HTTP Streaming* (*MJPEG*), która jest jednak **jednokierunkowa**. Aby zapewnić pełną interaktywność – w tym natychmiastowe sterowanie pozycją kamery (**PTZ**) oraz odbiór zdarzeń alarmowych bez *pollingu* – zdecydowano się na wykorzystanie protokołu **WebSocket**.

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

- `video_frame` / `audio_chunk`: Kanały **wysokiej przepustowości** do transmisji mediów.
- `motion_status`: Kanał **zdarzeń asynchronicznych** (*push notifications*) informujący o wykryciu ruchu.
- `control`: Kanał sterujący odbierający polecenia użytkownika (np. ruch silników PTZ).

Takie podejście wyeliminowało narzut związany z ciągłym nawiązywaniem nowych połączeń HTTP (*handshake*), co jest krytyczne dla zachowania **niskich opóźnień** (*low latency*).

Model współbieżności (*Concurrency Model*)

Serwer aplikacji został zaprojektowany w **modelu wielowątkowym** (*Multi-threaded*), 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 (*Shared State*)

Wyzwaniem w środowisku wielowątkowym jest bezpieczny dostęp do zasobów. W projekcie zastosowano **wzorzec Singleton** realizowany poprzez moduł `app/shared.py` (Listing 3.9). 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`.

Listing 3.9. Implementacja współdzielonego stanu aplikacji.

```
# app/shared.py
# Globalne instancje dostepne dla wszystkich watkow i handlerow
camera = None
video_streamer = None
audio_streamer = None
recorder = None
```

Listing 3.6: Moduł stanu wspoldzielonego app/shared.py

Źródło: Opracowanie własne.

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

3.5.9 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 płótnie (*canvas context*) metodą `drawImage()`.

Pominięcie bufora odtwarzacza wideo pozwoliło na uzyskanie efektu **czasu rzeczywistego** („Real-Time”), gdzie obraz widoczny na ekranie odpowiada aktualnemu stanowi sensora kamery.

Listing 3.10. Implementacja renderowania klatek na elemencie Canvas.

```
socket.on('video_frame', function(data) {
    const canvas = document.getElementById('video_canvas');
    const ctx = canvas.getContext('2d');

    const image = new Image();
    image.onload = function() {
        // Rysowanie klatki natychmiast po otrzymaniu
        ctx.drawImage(image, 0, 0, canvas.width, canvas.
           height);
    };
    image.src = '' + data.frame;
});
```

Listing 3.7: Renderowanie klatek wideo na Canvas

Źródło: Opracowanie własne.

Reprodukcja Dźwięku i Jitter Buffer

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

Kluczowym wyzwaniem inżynierskim była kompensacja **nierównomiernego dostarczania pakietów** przez sieć (tzw. *Network Jitter*). Bezpośrednie odtwarzanie próbek skutkowało słyszalnymi trzaskami i przerwami. Rozwiązaniem było zaimplementowanie programowego **bufora fluktuacji** (*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.

Listing 3.11. Implementacja kolejkowania audio z kompensacją jittera (Jitter Buffer).

```
// Stala opoznienia kompensacyjnego (50ms)
const JITTER_DELAY = 0.05;
let nextStartTime = 0;

socket.on('audio_chunk', function(data) {
    // 1. Konwersja surowych bajtow na Float32
    const int16Data = new Int16Array(data);
    const float32Data = new Float32Array(int16Data.length);
    for (let i = 0; i < int16Data.length; i++) {
        // Normalizacja do zakresu -1.0 do 1.0
        float32Data[i] = int16Data[i] / 32768;
    }

    // 2. Utworzenie bufora audio
    const audioBuffer = audioCtx.createBuffer(1, float32Data.
       length, 16000);
    audioBuffer.getChannelData(0).set(float32Data);

    // 3. Planowanie czasu odtworzenia (Scheduling)
    const source = audioCtx.createBufferSource();
    source.buffer = audioBuffer;
    source.connect(audioCtx.destination);

    // Algorytm Jitter Buffer:
    const now = audioCtx.currentTime;
    const playTime = Math.max(now + JITTER_DELAY,
       nextStartTime);

    source.start(playTime);
    nextStartTime = playTime + audioBuffer.duration;
```

```
29 });
```

Listing 3.8: Kolejkowanie audio z Jitter Buffer

Źródło: Opracowanie własne.

Interakcja i Sprzężenie Zwrotne (PTZ)

Interfejs sterowania kamerą (**PTZ**) zaprojektowano w oparciu o logikę „naciśnij i przytrzymaj", wykorzystując zdarzenia myszy `mousedown` i `mouseup/mouseleave`.

Zaimplementowano również **mechanizm wizualnego sprzężenia zwrotnego** dla stanów granicznych (*ptz_limit*). Serwer backendowy emituje zdarzenie, na które kod JavaScript reaguje, **dynamicznie blokując** (*wyszarzając*) odpowiedni przycisk kierunkowy, co informuje użytkownika o niemożności wykonania dalszego ruchu w danym kierunku.

3.5.10 Detekcja zdarzeń (Detekcja Ruchu)

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` (moduł `app/detection/motion.py`).

Algorytm adaptacyjnego modelowania tła

W przeciwieństwie 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**.

[Image of Frame Differencing vs Background Modeling in video surveillance]

Każda nowa klatka wpływa na wzorzec tła z określoną wagą α (w implementacji przyjęto $\alpha = 0.5$). 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** (np. zachód słońca), 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 gaussowskiemu** (`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), oznaczane są jako ruch (wartość 255).

4. **Ekstrakcja konturów:** Na obrazie binarnym wyszukiwane są ciągłe obszary zmian za pomocą funkcji `cv2.findContours`.

Listing 3.12. Implementacja algorytmu detekcji ruchu w klasie MotionDetector.

```
    def detect(self, frame):
        # 1. Pre-processing: Skala szarosci i rozmycie
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        gray = cv2.GaussianBlur(gray, (21, 21), 0)

        # Inicjalizacja modelu tla przy pierwszej klatce
        if self.avg is None:
            self.avg = gray.copy().astype("float")
            return False, frame

        # 2. Aktualizacja modelu tla (srednia wazona)
        cv2.accumulateWeighted(gray, self.avg, 0.5)

        # 3. Obliczenie roznicy (Delta)
        frameDelta = cv2.absdiff(gray, cv2.convertScaleAbs(
           self.avg))

        # 4. Binaryzacja i dylatacja (wypelnianie dziur)
        thresh = cv2.threshold(frameDelta, 25, 255, cv2.
           THRESH_BINARY)[1]
        thresh = cv2.dilate(thresh, None, iterations=2)
```

```
20
21          # 5. Znajdowanie konturow
22          cnts, _ = cv2.findContours(thresh.copy(), cv2.
               RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
23
24          motion_detected = False
25
26          # ... (Logika filtracji obszarowej)
```

Listing 3.9: Metoda detekcji ruchu wykorzystująca akumulację wagi

Źródło: Opracowanie własne.

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`.
- Wyrysowanie **ramki otaczającej** (*Bounding Box*) na klatce wynikowej, co stanowi wizualną informację dla operatora.

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

3.5.11 Moduł Rejestracji (Recorder)

Ostatnim ogniwem w łańcuchu przetwarzania danych jest moduł rejestracji, zaimplementowany w klasie `Recorder` (katalog `app/recording`). Jego zadaniem jest przechwycenie ulotnych strumieni wideo i audio oraz ich trwała archiwizacja w postaci **pliku multimedialnego** (kontener `MP4`). Ze względu na wymagania dotyczące wydajności czasu rzeczywistego, zaprojektowano go w oparciu o strategię **odroczonego zapisu** (*Deferred Writing*).

Strategia buforowania w pamięci (*In-Memory Buffering*)

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

drops).

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`).

Listing 3.13. Implementacja buforowania strumieni w pamięci RAM.

```
class Recorder:
    def __init__(self):
        self.is_recording = False
        self.video_frames = [] # Bufor wideo
        self.audio_chunks = [] # Bufor audio (surowe probki)
        # ...

    def start_recording(self):
        self.video_frames = []
        self.audio_chunks = []
        self.is_recording = True
        print("Recording Started")

    def add_frame(self, frame):
        if self.is_recording:
            # Konwersja BGR (OpenCV) -> RGB (MoviePy)
            frame_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB
                )
            self.video_frames.append(frame_rgb)

    def add_audio(self, audio_data):
        if self.is_recording:
            self.audio_chunks.append(audio_data)
```

Listing 3.10: Buforowanie strumieni w klasie Recorder

Źródło: Opracowanie własne.

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

Synteza pliku i Post-processing (MoviePy)

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.

3.5.12 Archiwizacja

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

Przetwarzanie wstępne i buforowanie (*Data Capturing*)

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 narzut obliczeniowy w trakcie nagrywania.

Finalizacja i synchronizacja A/V (*Stopping & Saving*)

Kluczowym wyzwaniem inżynierskim jest zapewnienie **synchronizacji obrazu z dźwiękiem** (tzw. *Lip-Sync*). W projekcie zastosowano metodę dynamicznego obliczania **rzeczywistego klatkażu** (*Real FPS*).

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:

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

Takie podejście **kompensuje** ewentualne wahania wydajności serwera (np. zgubione klatki w wyniku obciążenia CPU), gwarantując, że długość ścieżki wideo będzie idealnie dopasowana do długości ścieżki audio.

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, poddawane konwersji do formatu **stereo** oraz **upsamplingowi** do 44.1 kHz.
2. **Muksowanie (*Muxing*):** Strumień wideo i audio są scalane w kontenerze **MP4**.
3. **Persystencja:** Gotowy plik jest zapisywany w dedykowanym katalogu (np. `static/recordin`), co umożliwia jego natychmiastowe udostępnienie przez serwer WWW.

Ograniczenia implementacyjne (*RAM Management*)

Należy podkreślić, że przyjęta strategia ***In-Memory Buffering*** (buforowanie 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. *clips*), typowych dla systemów detekcji ruchu.

3.5.13 Konteneryzacja i Wdrożenie (Docker)

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. Definicję obrazu zawarto w pliku `Dockerfile`.

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 artefaktu oraz zmniejszenia **powierzchni ataku** (*security attack surface*).

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.
- `libgl1` / `libglib2.0-0`: Biblioteki graficzne wymagane przez `opencv-python-headless` do operacji na macierzach obrazu.

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

Listing 3.15. Definicja środowiska uruchomieniowego w pliku Dockerfile.

```
FROM python:3.13-slim

# Instalacja zaleznosci systemowych dla OpenCV i FFmpeg
RUN apt-get update && apt-get install -y \
    ffmpeg \
    libgl1-mesa-glx \
    libglib2.0-0 \
    && rm -rf /var/lib/apt/lists/*

# Kopiowanie menedzera pakietow uv (z wczesniej zbudowanej warstwy)
COPY --from=ghcr.io/astral-sh/uv:latest /uv /bin/uv

# Instalacja zaleznosci Python
COPY pyproject.toml uv.lock /app/
WORKDIR /app
RUN uv sync --frozen

# Kopiowanie kodu aplikacji
COPY . /app

# Uruchomienie serwera
CMD ["uv", "run", "python", "run.py"]
```

Listing 3.11: Definicja obrazu Dockerfile z instalacją zależności systemowych

Źródło: Opracowanie własne.

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** (*Environment Variables*). Jest to zgodne z metodyką *Twelve-Factor App*.

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 (*Fallback*).

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

Listing 3.16. Hybrydowy mechanizm ładowania konfiguracji (Env Vars > JSON).

```
def load_config():
    # 1. Proba pobrania konfiguracji ze zmiennych
       srodowiskowych (Docker)
    if os.environ.get("TAPO_IP"):
        return {
            "host": os.environ.get("TAPO_IP"),
            "user": os.environ.get("TAPO_USER"),
            "password": os.environ.get("TAPO_PASSWORD"),
            # ...
        }

    # 2. Fallback do pliku lokalnego (Development)
    try:
        with open('config.json', 'r') as f:
            return json.load(f)
    except FileNotFoundError:
        return None
```

Listing 3.12: Ladowanie konfiguracji z priorytetyzacja zmiennych srodowiskowych

Źródło: Opracowanie własne.

3.6 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 opisanym w tej części pracy jest **skuteczna integracja warstwy sprzętowej** kamery Tapo C200 z aplikacją webową, pomimo ograniczeń narzucanych 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 multimediów (*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 programistycznego, gotowego do empirycznej weryfikacji. Kolejny rozdział poświęcony zostanie **testom wydajnościowym** oraz analizie jakościowej tak przygotowanego rozwiązania.

4 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 skuteczną i bezpieczną alternatywę dla zamkniętego ekosystemu producenta.

4.1 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 wewnątrz odizolowanej sieci lokalnej.

Infrastruktura sprzętowa i systemowa (Host)

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

Tabela 4.1: 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 rozdzielczości **Full HD (1080p)** przy 30 FPS. Rozwiązanie 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.

4.2 Przebieg scenariuszy testowych

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

4.2.1 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 rozdzielczości 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 rozdzielczości **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 FPS**.

4.2.2 Test T02: Stabilność i płynność strumieniowania wideo

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 wyrenderowanej 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 występowały 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.

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.
- **Zmienność 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.

4.2.3 Test T03: Wpływ procesu rejestracji na zasoby pamięci operacyjnej

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 rozwiązaniu, 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:

$$M(t) = a \cdot t \tag{1}$$

Gdzie:

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

4.3 Wnioski i Analiza

4.3.1 Analiza wydajności algorytmu detekcji ruchu (Wnioski z testu T01)

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 nadrzędny 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 wewnątrz 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 kompilowanego, takiego jak C++.

4.3.2 Analiza stabilności i płynności strumieniowania wideo (Wnioski z testu T02)

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 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 wewnętrznego 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 kodeka H.264 Zaobserwowane podczas testów artefakty wizualne (błędy dekodowania) potwierdziły specyfikę pracy z protokołem UDP w środowisku sieci bezprzewodowych. Choć protokół ten zapewnia szybszą transmisję niż TCP, brak mechanizmu retransmisji pakietów w połączeniu z charakterystyką kodeka 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.

4.3.3 Analiza wpływu procesu rejestracji na zasoby (Wnioski z testu T03)

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 Zastosowana 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).

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 MB/s (dla nieskompresowanych klatek 1080p).
- **Rzeczywiste zużycie:** $\approx$148 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 minuty 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*).

4.4 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 rozwiązaniach **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, rozwiązując 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 1920×1080 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 kompilowanego (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. potokowanie 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 produk-

cyjnych rekomendowane jest zastosowanie połączenia przewodowego (Ethernet) lub wdrożenie programowego bufora korekcyjnego (ang. *jitter buffer*), co jednak wiązałoby się z kompromisem w postaci zwiększonego opóźnienia.

Wnioski Końcowe

Kierunki dalszego rozwoju

Podsumowanie pracy

Bibliografia

Antonakakis, Manos i in. (2017). „Understanding the Mirai Botnet". W: *26th USENIX Security Symposium (USENIX Security 17)*, s. 1093–1110.

Gonzalez, Rafael C i Richard E Woods (2018). *Digital Image Processing*. 4 wyd. Pearson.

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.

Mi, Xianghang i in. (2019). „Resident Evil: Understanding Residential IP Proxy as a Dark Service". W: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, s. 1185–1201.

Neshenko, Nataliia i in. (2019). „Vulnerability Analysis of Low-Cost Consumer IoT Devices". W: *IEEE Communications Surveys & Tutorials* 21.4, s. 3236–3273.

Opara-Martins, Justice, Reza Sahandi i Feng Tian (2016). „A decision-making framework for cloud computing migration to avoid vendor lock-in". W: *Journal of Cloud Computing* 5.1, s. 1–15.

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.

Sullivan, Gary J i in. (2012). „Overview of the high efficiency video coding (HEVC) standard". W: *IEEE Transactions on circuits and systems for video technology* 22.12, s. 1649–1668.

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

Wiegand, Thomas i in. (2003). „Overview of the H. 264/AVC video coding standard". W: *IEEE Transactions on circuits and systems for video technology* 13.7, s. 560–576.

;

Spis rysunków

Spis tabel