POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: INFORMATYKA
SPECJALNOŚĆ: SYSTEMY INFORMATYKI W MEDYCYNIE

PRACA DYPLOMOWA INŻYNIERSKA

System inspekcji obszarów z wykorzystaniem autonomicznych dronów

Autonomous drone-based scouting system

AUTOR:

Mateusz Bączek

PROWADZĄCY PRACĘ:

Dr inż. Michał Kucharzak, Katedra Systemów i Sieci Komputerowych

WROCŁAW, 2020

Spis treści

Skróty

JSON (ang. *JavaScript Object Notation*) – format serializacji danych, natywny dla języka *JavaScript*, wykorzystywany głównie w technologiach webowych.

CI/CD (ang. *Continuous Improvement/Continuous Delivery*) – zestaw praktyk i narzędzi, wykorzystywanych w celu automatyzacji procesu testowania, ulepszania i wdrażania oprogramowania.

Obraz dockera (ang. *Docker image*) – bazowy obraz systemu plików, na którym zainstalowana jest dana aplikacja (lub wiele aplikacji).

Kontener dockera (ang. *Docker container*) – obraz dockera, który został **uruchomiony** na komputerze. Posiada własny stan – uruchomione procesy oraz zmiany w systemie plików.

Rozdział 1

Wstęp

1.1. Geneza pracy

Sektor gospodarki związany z inżynierią bezzałogowych maszyn lotniczych znajduje się w stanie dynamicznego rozwoju i generuje coraz większe przychody. Tym samym przyciąga inwestorów skłonnych zainwestować w innowacyjne pomysły zarówno dużych przedsiębiorstw, jak i młodych konstruktorów. Międzynarodowe zawody sponsorowane przez podmioty prywatne i organizacje rządowe umożliwiają pasjonatom wdrożenie w życie ich wizji. Przykład stanowi *UAV Challenge* – zawody skoncentrowane na autonomicznych systemach wsparcia służb medycznych, współorganizowane przez rząd australijskiego stanu Queensland, które przyciągają corocznie najlepsze zespoły z całego świata[1]. Potencjał inwestycyjny i naukowy, którym dysponuje ta branża, sprzyja powstawaniu różnorodnych rozwiązań. W konsekwencji bezpośrednio niezwiązane z lotnictwem sektory korzystają z dorobku autonomicznej awiacji. Tytułem przykładu: generowanie trójwymiarowych map terenu, przy użyciu zdjęć wykonanych z dronów znajduje zastostowanie w dziedzinach takich jak górnictwo, geodezja czy militaria [2], natomiast dostarczanie towarów indywidualnym klientom z pomocą dronów jest w stanie zrewolucjonizować przemysł transportowy [3].

System realizowany w ramach pracy mógłby potencjalnie służyć jako wsparcie dla służb medycznych, wspomagając poszukiwanie osób zaginionych. Jest jednak to jedynie pojedynczy przykład z szerokiego spektrum potencjalnych zastosowań. Algorytmy sztucznej inteligencji, zastosowane do analizy zdjęć wykonanych w czasie lotów, mogą zostać wytrenowane na nowych danych, aby rozpoznawać dowolne obiekty (samochody, budynki, drzewa). Dzięki temu, system może zostać szybko przystosowany do nowych celów.

Budowa systemu wykorzystującego fizyczne komponenty, takie jak autonomiczne drony, wymaga wprowadzenia dodatkowych zabezpieczeń, pozwalających na przetestowanie stabilności systemu, zanim zostanie uruchomiony w terenie. Wymaganie podyktowane jest wysokim kosztem dronów oraz względami bezpieczeństwa – utrata kontroli nad maszyną latającą, gdy wokół znajdują się ludzie, stanowi duże zagrożenie. Nowoczesne praktyki inżynierii oprogramowania,

takie jak automatyczne testy w potoku *CI/CD* oraz zastosowanie symulatorów i atrap w miejsce rzeczywistych, fizycznych obiektów pozwalają na budowę takich zabezpieczeń.

Przedstawiony w pracy prototyp systemu inspekcji obszarów powstał w ramach działalności Akademickiego Klubu Lotniczego – koła naukowego Politechniki Wrocławskiej, zajmującego się rozwijaniem technologii związanych z autonomicznymi dronami i samolotami [4]. Projekt został sfinansowany przez Wrocławski Fundusz Aktywności Studenckiej [5].

1.2. Cel pracy

Celem pracy jest zaprojektowanie architektury i budowa prototypu systemu inspekcji obszarów, wykorzystującego autonomiczne drony. System ma wykorzystywać napisaną w ramach pracy infrastrukturę informatyczną, pozwalającą na planowanie tras lotów, obsługę telemetrii i rozpoznawanie obiektów na zdjęciach wykonanych w czasie lotu, za pomocą algorytmów sztucznej inteligencji.

Architektura systemu (wykorzystane technologie i struktura podprojektów składających się na system) musi pozwalać na zautomatyzowanie procesu testowania i wdrażania systemu – każde z wdrożeń musi być poprzedzone testami integracyjnymi na poziomie całego systemu. Aby przeprowadzić pełne testy integracyjne, prototyp systemu będzie uruchamiany w środowisku testowym, zawierającym zintegrowany symulator drona/samolotu.

Finalnie, system ma pozwolić użytkownikowi na zaplanowanie trasy przelotu drona oraz wyznaczenie harmonogramu, według którego mają odbywać się loty. W trakcie lotu, system ma w czasie rzeczywistym dostarczać użytkownikowi dane telemetryczne o obecnej pozycji maszyny i pozwalać na podgląd zdjęć wykonywanych w czasie lotu. Po wylądowaniu, system wygeneruje dla użytkownika podsumowanie lotu, zawierające trasę pokonaną przez drona, wraz z wykonanymi zdjęciami, na których oznaczone zostaną rozpoznane przez algorytmy sztucznej inteligencji obiekty.

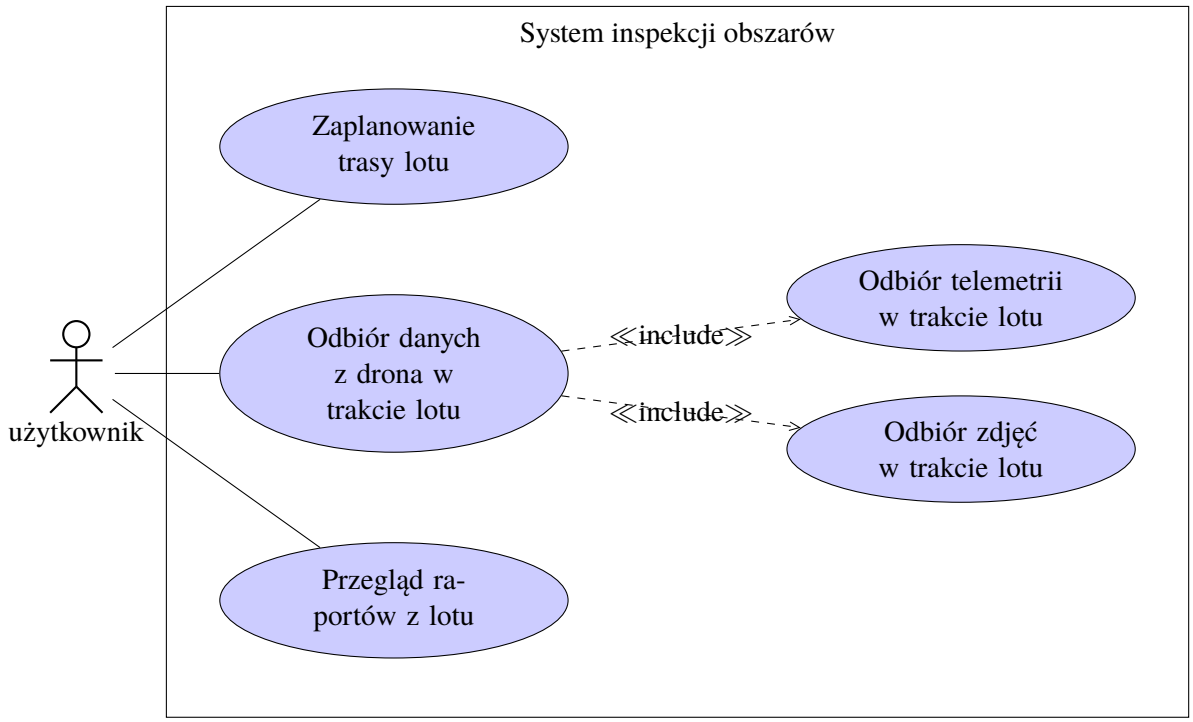
1.3. Zakres i struktura pracy

Zakres pracy obejmuje elementy projektu związane z inżynierią i architekturą oprogramowania oraz proces budowy i testowania prototypu systemu. Poszczególne rozdziały pracy skupione są wokół różnych zadań, realizowanych w ramach projektu:

- zdefiniowanie wymagań funkcjonalnych systemu – rozdział 2,
- projekt architektury systemu – rozdział 3,
- opis zastosowanej metodyki wdrażania i jej wpływ na rozwój projektu – rozdział 4.
- zaprojektowanie i przeprowadzenie testów – rozdział 5.

Rozdział 2

Wymagania funkcjonalne systemu



Rys. 2.1: Diagram przypadków użycia systemu

Rysunek 2.1 ilustruje przypadki użycia systemu, opisane w celu pracy (1.2). Poniższy rozdział opisuje, jak przypadki użycia przekładają się na wymagania funkcjonalne systemu.

2.1. Oprogramowanie na dronie

Wielowirnikowce podłączone do systemu muszą być zdolne do autonomicznego lotu – w kontekście pracy oznacza to zdolność do automatycznego startu, lądowania, stabilizacji oraz samodzielnego lotu do koordynatów GPS. W trakcie lotu, maszyna musi zbierać i wysyłać dwa rodzaje danych:

- dane telemetryczne,

- zdjęcia wykonane w czasie lotu.

Dane telemetryczne muszą zawierać informacje o pozycji drona oraz identyfikować maszynę za pomocą unikatowego identyfikatora oraz numeru lotu. Przesyłany jest też poziom naładowania baterii oraz informacja, czy w obecnym czasie prowadzone jest nagrywanie.

2.2. Protokoły wymiany danych

W przypadku systemu działającego autonomicznie, wymiana danych jest kluczowym elementem pozwalającym na sprawdzanie poprawności działania i diagnozowania błędów w systemie. Podczas lotów testowych często nie ma możliwości bezpośredniej obserwacji systemu lub ingerencji w jego sposób działania. Odpowiednia architektura zbierająca i archiwizująca dane z lotów pozwala znacznie szybciej wykryć potencjalne problemy i zapobiec krytycznym błędom.

2.2.1. Dane telemetryczne

Protokół do wymiany danych telemetrycznych powinien wysyłać dane w postaci binarnej, gdyż jest to bardziej efektywne niż kodowanie ich w postaci tekstowej (na przykład w formacie `JSON`, typowym dla języków wysokopoziomowych – wykorzystywanym w technologiach webowych).

Protokół powinien być w łatwy sposób rozszerzalny, pozwalając w przyszłości zredefiniować część wysyłanych pakietów lub dodać nowe dane, bez tracenia kompatybilności wstecznej bądź konieczności przebudowania całego systemu. Poszczególne komponenty systemu będą pisane w różnych językach programowania – biorąc to pod uwagę, pożądaną cechą protokołu jest też możliwość szybkiego przeportowania go na inny język programowania.

2.2.2. Zdjęcia wykonane w trakcie lotu

Efektywny przesył zdjęć oraz filmów to temat zbyt obszerny i wymagający, żeby poruszać go w treści pracy – system powinien wykorzystywać dowolny prosty w implementacji protokół wysyłania zdjęć. Architektura systemu powinna zapewnić możliwość prostej wymiany tego komponentu, dzięki czemu w przyszłości będzie możliwe zastąpienie go przez bardziej zoptymalizowane rozwiązanie.

2.3. Oprogramowanie serwerowe

Serwer webowy jest komponentem, który odbiera, archiwizuje i przekazuje dane nadchodzące z dronów do aplikacji klienckiej. Jest punktem centralnym systemu, wykorzystywanym bezpośrednio przez wszystkie pozostałe elementy.

2.3.1. Odbiór i multipleksowanie telemetrii

Aby umożliwić diagnozowanie stanu systemu w czaie rzeczywistym – szczególnie stanu wykonujących lot wielowirnikowców, telemetria nadchodząca z maszyn nie może być jedynie archiwizowana na dysku serwera centralnego. Konieczną funkcjonalnością jest przesyłanie jej w czasie rzeczywistym do wielu jednocześnie podłączonych klientów.

Umożliwi to podjęcie akcji w przypadku wykrycia krytycznego błędu, który mógłby zakończyć się uszkodzeniem bądź rozbiciem drona, ułatwi też wykonywanie testów – zarówno na rzeczywistych maszynach, jak i wykorzystujących symulatory lotu.

2.4. Oprogramowanie klienckie

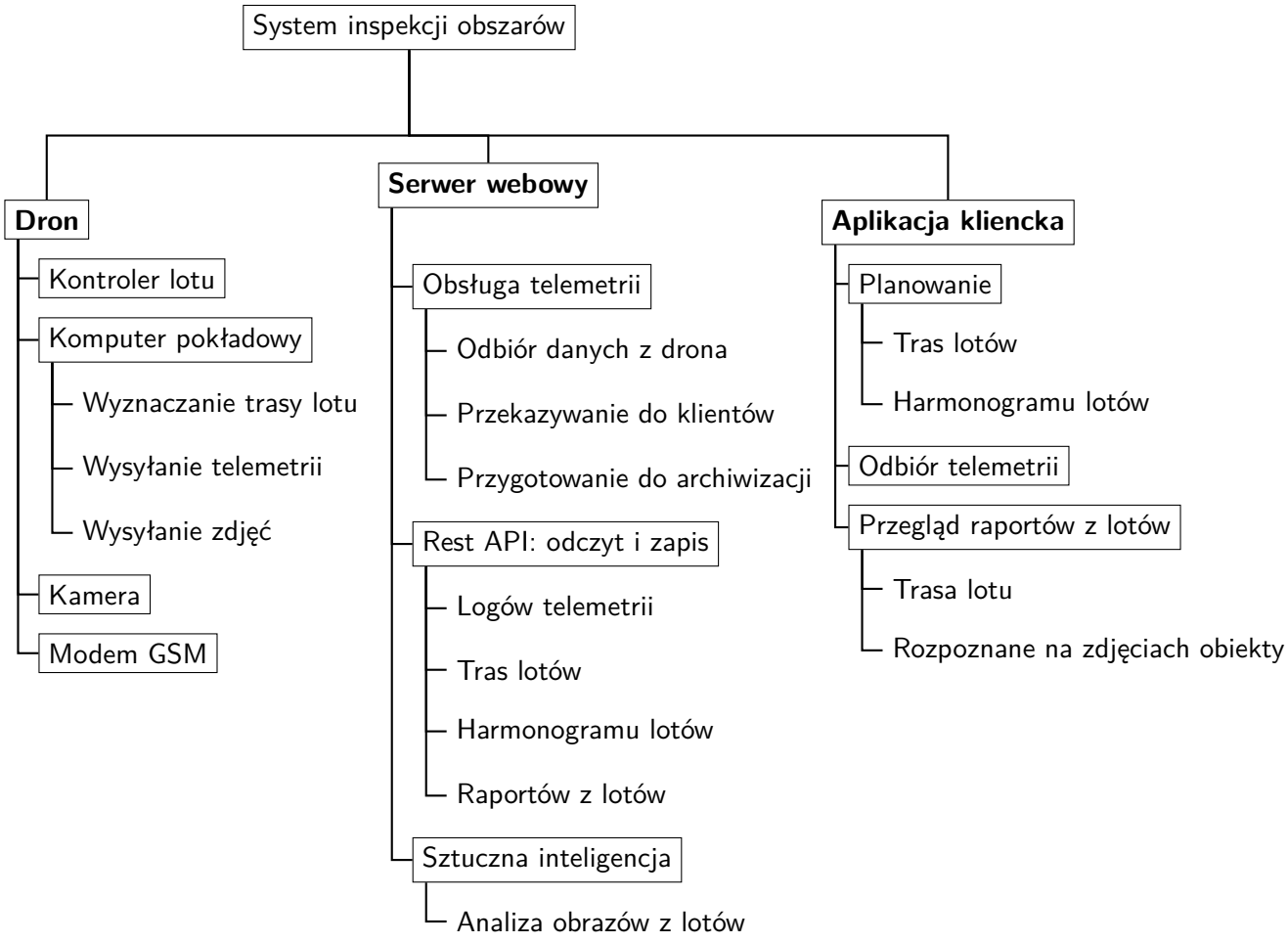
Aplikacja kliencka skupiona jest wokół trzech funkcjonalności:

1. planowanie tras i harmonogramu lotów,
2. odbiór telemetrii i zdjęć z dronów w czasie rzeczywistym,
3. przegląd i analiza telemetrii oraz zdjęć zarchiwizowanych z poprzednich lotów.

Kluczowym elementem aplikacji klienckiej jest obsługa mapy – wszystkie wymienione funkcjonalności wymagają wizualizacji nadchodzących danych geograficznych, rozszerzonych o dodatkowe informacje (na przykład godzina przelotu przez dany punkt lub zarejestrowane w danym miejscu zdjęcia i wykryte na nich obiekty).

Rozdział 3

Architektura systemu: przegląd i wybór technologii



Rys. 3.1: Zarys architektury systemu

Diagram 3.1 opisuje architekturę systemu, wynikającą z wymagań funkcjonalnych opisanych w rozdziale 2. Poniższy rozdział opisuje proces wyboru technologii, dzieki którym będzie możliwa budowa systemu spełniającego te wymagania.

3.1. Dron

3.1.1. Kontroler lotu

Wielowirnikowce podłączone do systemu, muszą być wyposażone w kontroler lotu – umożliwiający autonomiczny lot, stabilizację oraz obsługę peryferiów takich jak czujniki oraz silniki.

Spośród aktywnie rozwijanych i popularnych projektów [6] tworzących oprogramowanie do kontrolerów lotu, można wyróżnić cztery najpopularniejsze inicjatywy:

Tab. 3.1: Najpopularniejsze otwarte projekty oprogramowania obsługującego kontrolery lotu

Nazwa projektu	**Rok założenia**	**Docelowy hardware**	**Licencja**
ArduPilot[7]	2009	otwarte mikrokontrolery ARM	GPLv3
AutoQuad[8]	2011	mikrokontrolery STM Cortex M4	GPLv3
LibrePilot[9]	2015	zamknięte źródłowo kontrolery lotu, bazujące na architekturze ARM	GPLv3
PX4 Autopilot[10]	2012	otwarte mikrokontrolery ARM	BSD

Spośród wymienionych projektów, ArduPilot posiada najbardziej rozbudowaną bazę dokumentacji i instrukcji. Architektura projektu umożliwia skompilowanie projektu na standardowy komputer typu PC uruchomienie go w wirtualnym środowisku[11], co ułatwia proces testowania oprogramowania, które steruje dronem – model maszyny jest symulowany, jednak warstwa komunikacji jest dokładnie taka sama, jak w przypadku pracy z prawdziwym dronem. Dodatkowo, projekt realizowany był w kole studenckim, w którym ArduPilot jest od lat wykorzystywany do sterowania dronami i samolotami, co przesądziło o zastosowaniu go jako oprogramowanie do kontrolera lotu.

3.1.2. Komputer pokładowy

Poza kontrolerem lotu, który zawiera jedynie oprogramowanie niezbędnie do sterowania dronem i udostępniania strumienia telemetriii, na maszynie musi znaleźć się też komputer pokładowy, do zastosowań bardziej ogólnych. Będzie on wykorzystywany do obsługi wysokopoziomowych peryferiów: kamery oraz modemu GSM. Do zadań komputera pokładowego będzie należała także realizacja logiki biznesowej systemu – odczytanie harmonogramu przelotów z serwera webowego i załadowanie do kontrolera lotu konkretnej trasy przelotu.

Wymagania sprzętowe

Aby wypełniać zadania wymienione powyżej, komputer pokładowy musi posiadać następujące interfejsy sprzętowe:

- `UART` – do komunikacji z kontrolerem lotu,
- `USB` – do komunikacji z modemem,
- `CSI` – do komunikacji z kamerą.

Większość dostępnych na rynku komputerów klasy SBC (*Single-board Computer*) posiada powyższe interfejsy, więc wymagania sprzętowe nie są tutaj ograniczeniem. W projekcie został wykorzystany najpopularniejszy do zastosowań amatorskich komputer *Raspberry Pi*.

Oprogramowanie

Na komputerze pokładowym zainstalowany jest system Linux – gwarantuje to bezproblemową obsługę peryferiów oraz dostępność stosu sieciowego, koniecznego do przesyłania danych telemetrycznych i zdjęć. Dodatkowo, wymagane jest oprogramowanie dekodujące i enkodujące wiadomości protokołu *MavLink*, który jest wykorzystywany przez ArduPilota do komunikacji z zewnętrznymi systemami [12].

Infrastruktura projektu ArduPilot dostarcza gotowe narzędziae do parsowania i tworzenia wiadomości w protokole *MavLink*. Jedną z nich jest `pymavlink` – implementacja protokołu *MavLink* w języku Python. `pymavlink` zawiera podstawowe funkcje i obiekty konieczne do komunikacji z kontrolerem lotu. Biblioteka jest niskopoziomowa i nie w całości napisana w sposób obiektowy – finalnie wykorzystujemy więc bibliotekę `dronekit-python`[13], która rozbudowuje `pymavlink` o w pełni obiektowy, wysokopoziomowy interfejs do komunikacji z kontrolerem lotu. Skrypty odpowiedzialne za komunikację z kontrolerem lotu i z serwerem webowym pisane są *Pythonie*.

3.2. Protokoły wymiany danych

3.2.1. Dane telemetryczne – biblioteka `protobuf`

W trakcie lotu, drony regularnie wysyłają dane telemetryczne. Każdy pakiet danych telemetrycznych w systemie zawiera następujące informacje:

- unikatowy identyfikator maszyny,
- identyfikator lotu (unikatowy dla każdej maszyny),
- pozycję odczytaną z odbiornika GPS,
- poziom naładowania baterii,
- informacja czy w obecnej chwili wykonywane są zdjęcia z lotu,
- timestamp w standardowym 32-bitowym formacie unixowym.

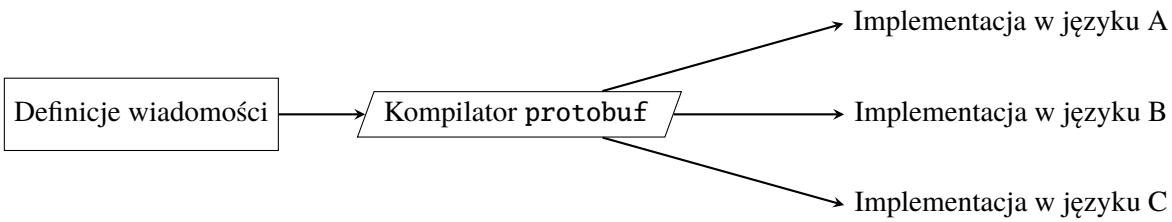
Nie jest wykluczone, że w przyszłości może pojawić się potrzeba dołączenia do danych telemetrycznych nowych informacji, na przykład w przypadku, gdy do maszyny zostanie dodane nowe oprzyrządowanie, lub gdy system byłby adaptowany do obsługi innego rodzaju misji. Dodatkowo, skala projektu wymusza utrzymywanie implementacji tego samego protokołu w wielu różnych językach programowania – oprogramowanie wysyłające telemetrię z drona napisane jest w Pythonie, interfejs webowy wyświetlający telemetrię będzie musiał być jednak napisany w języku JavaScript (ponieważ tylko ten język jest wspierany przez przeglądarki internetowe). W przyszłości może pojawić się konieczność dodania wsparcia dla innego języka programowania – na przykład gdyby do systemu miałby być dodany kolejny komponent. Jest to poten-

cjalne źródło problemów związanych z integracją między komponentami systemu, ponieważ wraz z dodawaniem kolejnych danych telemetrycznych i zwiększaniem liczby wspieranych języków, rośnie szansa na popełnienie błędu w implementacji protokołu. Nawet w przypadku dwóch równolegle wspieranych języków, utrzymywanie spójnej implementacji protokołu nie jest zadaniem trywialnym – szczególnie jeśli protokół wysyła dane w formie binarnej, co jest konieczne, gdy jednym z wymagań jest wydajne zapakowanie danych.

Jednym z możliwych rozwiązań tego problemu jest wykorzystanie narzędzi do generowania kodu. Narzędzia tego typu pozwalają na zdefiniowanie standardu danych telemetrycznych we własnym (specyficznym dla narzędzia) języku, opisującym strukturę danych przesyłanych przez protokół. Następnie, narzędzia generują implementację protokołu w wybranych językach programowania (jak zilustrowano na diagramie 3.2). Dzięki automatycznemu generowaniu implementacji, programiści nie muszą utrzymywać tego samego protokołu w kilku różnych językach programowania.

Wykorzystywanym w projekcie narzędziem do generowania implementacji protokołu jest `protobuf` (*Protocol Buffers*)[14] – biblioteka napisana w firmie Google, stworzona z myślą o zapewnianiu wydajnej komunikacji w czasie rzeczywistym pomiędzy wieloma systemami informatycznymi. W obecnej wersji (`v3.14.0`), biblioteka pozwala na generowanie kodu w językach:

- Python
- C++
- Java
- C#
- JavaScript
- Go



Rys. 3.2: Diagram przedstawiający system generowania implementacji protokołu, na podstawie narzędzia `protobuf`

Poza automatycznym generowaniem kodu protokołu, `protobuf` zapewnia także:

- wydajne wykorzystanie miejsca – biblioteka upakowuje dane w formie binarnej,
- kompatybilność wsteczną – w przypadku dodania nowego typu pakietu telemetrii, implementacja zachowuje spójność z poprzednimi wersjami. Dzięki temu możliwe jest stopniowe wprowadzanie zmian w wielokomponentowym systemie,
- automatycznie generowane mechanizmy, pozwalające na sprawdzanie, czy dane umieszczone w pakiecie telemetrii są poprawne (sprawdzanie typów, sprawdzanie czy zostały wypełnione wszystkie pola pakietu),

- interfejs programistyczny, umożliwiający samodzielne dopisanie wsparcia nowych języków do kompilatora `protobuf`.

Listing 3.1: Przykład definicji pakietu `protobuf`

```
syntax = "proto3";

package Telemetry;

/**
  Wiadomość zawierająca pozycję
  maszyny w trakcie lotu
*/
message Position {
  float lat = 1;
  float lng = 2;
  float alt = 3;
  float heading = 4;
}

message TelemFrameHeader {
  /* Flight metadata */
  int32 timestamp = 1;
  int32 machine_id = 2;
  int32 flight_id = 3;

  /**
  Kompozycja wcześniej
  zdefiniowanej wiadomości
  */
  Position position = 4;
}
```

Listing 3.2: Przykład wykorzystania wygenerowanej implementacji pakietów w języku Python

```
# Wygenerowany moduł, zawierający definicje
# pakietów, domyślnie nosi nazwę definitions_pb2
import definitions_pb2
import time

header = definitions_pb2.TelemFrameHeader()

header.timestamp = int(time.time())
header.machine_id = 1
header.flight_id = 5

header.position.lat = 5
header.position.lng = 10
header.position.alt = 15
header.position.heading = 20
```

```
16
17 # Postać binarna, gotowa do zapisania do pliku
18 # lub wysłania przez protokół UDP/WebSocket
19 serialised_header = header.SerializeToString()
20 print("Serialised header:")
21 print(serialised_header)
```

3.2.2. Zdjęcia wysyłane w trakcie lotu – biblioteka `imagezmq`

Wykonane przez drony zdjęcia wysyłane są na serwer webowyza pomocą biblioteki `imagezmq`. Jak zaznaczono w podrozdziale wymagań funkcjonalnych, poświęconym wysyłaniu zdjęć z lotów (2.2.2), wysyłanie strumienia wideo to temat zbyt skomplikowany, aby poruszać go w pracy – rozwiązanie wykorzystywane do przesyłu zdjęć zostało wybrane ze względu na prostotę instalacji i implementacji.

Biblioteka `imagezmq` pozwala wysyłać obrazy za pomocą protokołu `zmq`. Przed wysłaniem, obrazy są kompresowane, jednak jest to kompresja ograniczająca się do pojedynczego kadru – nie zaś strumienia obrazów, jak by to miało miejsce w przypadku strumieniowania filmu z lotu.

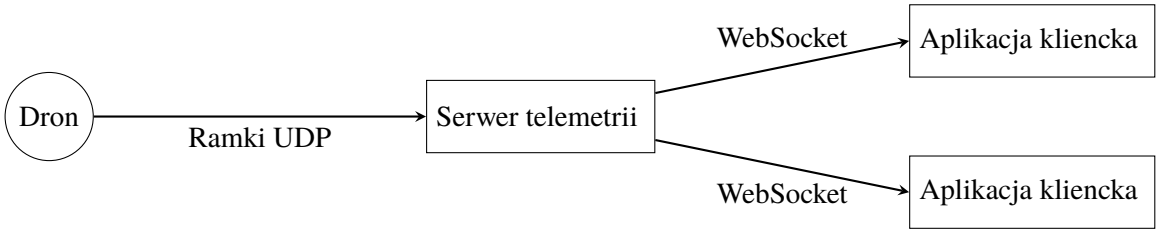
3.3. Oprogramowanie serwerowe

Serwer webowy jest punktem stykowym wszystkich elementów w systemie. Na serwerze przechowywane są wszystkie dane konieczne do działania systemu, takie jak trasy i harmonogramy lotów. Nadchodząca z dronów telemetria jest na serwerze multipleksowana i archiwizowana, pozwalając zarówno na odbieranie jej w czasie rzeczywistym z poziomu aplikacji klienckiej, jak i na analizę już zakończonych lotów.

3.3.1. Obsługa telemetrii

System obsługujący dane telemetryczne odbiera nadchodzące z maszyn pakiety `protobuf` (opisane w podrozdziale 3.2.1), następnie w czasie rzeczywistym przekazuje je do aplikacji klienckich – w ten sposób możliwy jest jednoczesny odbiór telemetrii na dowolnie wielu urządzeniach. Poszczególne loty rozróżnialne są za pomocą unikatowej krotki `(id_maszyny, id_lotu)`. Gdy system zauważy, że pakiety telemetryczne z danego lotu przestały napływać (przez ponad 10 sekund nie pojawił się żaden nowy pakiet o danym `(id_maszyny, id_lotu)`), archiwizuje otrzymane w ramach tego lotu pakiety, aby możliwe było przeanalizowanie ich po locie.

Rys. 3.3: Architektura systemu obsługującego telemetrię

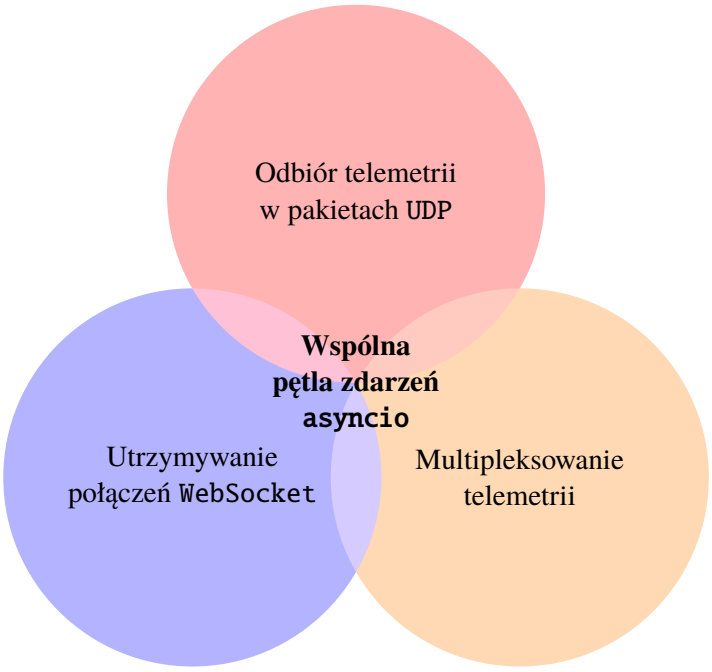


Pakiety pomiędzy dronem a serwerem telemetrii przesyłane są za pośrednictwem protokołu UDP. Został wybrany, ponieważ jest protokołem bezpołączeniowym – będzie odporny na potencjalne straty zasięgu, które mogą wystąpić w przypadku przelotu maszyny przez miejsce o słabym pokryciu siecią GSM. Wykorzystanie protokołu UDP zapewnia też mniejszy narzut dodatkowych danych, niż w przypadku wykorzystania TCP.

Po odebraniu ramek UDP na serwerze telemetrii, dane telemetryczne rozsyłane są do wszystkich podłączonych obecnie aplikacji klienckich. Aplikacja kliencka może być uruchomiona na dowolnej przeglądarce internetowej, w związku z czym wykorzystanie niskopoziomowych protokołów nie jest tutaj możliwe – przeglądarki internetowe nie obsługują ich ze względów bezpieczeństwa. Nadal wymagana jest jednak komunikacja w czasie rzeczywistym. Aby spełnić te wymagania, wykorzystany został protokół `WebSocket`[15], obsługiwany przez wszystkie powszechnie używane przeglądarki internetowe.

System obsługujący telemetrię będzie musiał równolegle obsługiwać nadchodzące ramki protokołu `UDP` oraz aplikacje klienckie, połączone za pośrednictwem protokołu `WebSocket`, uniemożliwia wykorzystanie wysokopoziomowego frameworka, napisanego z myślą o typowych projektach w obrębie technologii webowych – tego typu narzędzia nie zapewniają wsparcia dla niskopoziomowych protokołów, takich jak UDP.

W projekcie wykorzystana została biblioteka `asyncio` – służąca między innymi do pisania kodu odpowiedzialnego za komunikację sieciową[16]. `Asyncio` pozwala na łatwe zinterfejsowanie ze sobą asynchronicznie działających korutyn, jednocześnie wykonujących różne operacje sieciowe. Diagram 3.4 ilustruje zadania wykonywane przez serwer telemetrii, za pośrednictwem `asyncio`.



Rys. 3.4: Zadania wykonywane równolegle wewnątrz systemu obsługi telemetrii. Wspólny interfejs obsługi korutyn oraz wymianę danych pomiędzy zadaniami zapewnia biblioteka `asyncio`.

3.3.2. Rest API

System zaopatrzony jest w standardowy serwer `Rest API`[17], pozwalający na zapis i odczyt:

- Tras lotów
- Harmonogramów lotów
- Danych telemetrycznych, odebranych w trakcie lotu
- Obrazów otrzymanych w trakcie lotu

Wszystkie obecnie używane popularne języki programowania, posiadają przynajmniej jeden aktywnie rozwijany framework służący do pisania interfejsów `Rest API`. W projekcie wykorzystano język Python i framework `Django`, jednak równie dobrze sprawdzi się tutaj dowolna inna technologia (`Spring`, `.NET`, `Ruby on Rails`).

3.3.3. Sztuczna inteligencja – analiza obrazów z lotów

Ostatnim komponentem tworzącym oprogramowanie serwerowe jest moduł sztucznej inteligencji, rozpoznający obiekty na zdjęciach wykonanych w trakcie lotu. Zakres pracy nie obejmuje tematów związanych ze szczegółami działania sztucznej inteligencji – implementacja ogranicza się do wykorzystania gotowego rozwiązania i zintegrowania go z pozostałymi komponentami systemu (w szczególności z systemem odbierającym zdjęcia z lotu).

Do rozpoznawania obrazów wykorzystany został otwartoźródłowy system YOLO – *You Only Look Once* [18]. W ramach integracji, system został skompilowany do formy biblioteki współdzielonej i podłączony do skryptu w języku *Python*. Skrypt monitoruje określony folder w systemie plików i rozpoznaje zdjęcia, które się w nim pojawią. Na zdjęciu 3.5 przedstawiono przykładowy rezultat działania systemu rozpoznawania ludzi na zdjęciach.

Rys. 3.5: Zdjęcie wykonane w czasie lotów testowych. System rozpoznał na nim 2 osoby.

3.4. Oprogramowanie klienckie

Aplikacja kliencka pozwala na wykonanie następujących działań:

- zaplanowanie przelotu:
  - zaplanowanie trasy,
  - wybór prędkości, z jaką mają być pokonywane konkretne segmenty trasy,
  - wybór segmentów trasy, podczas których mają być wykonywane zdjęcia,
  - zaplanowanie godziny wylotu.
- podgląd telemetrii w czasie lotu,
- przegląd danych teletrycznych zapisanych z poprzednich lotów:
  - podlgąd trasy pokonanej przez maszynę,
  - podgląd wykonanych przez maszynę zdjęć, wraz z informacjami o rozpoznanych obiektach,

Najbardziej istotnym z wymagań jest obsługa mapy, która będzie obecna we wszystkich widokach aplikacji. W systemie została wykorzystana biblioteka `Leaflet`, służąca do wizualizowania danych na mapach. Rysunek 3.6 zawiera jeden z widoków finalnej wersji aplikacji klienckiej, wykorzystującej bibliotekę `leaflet` w komponencie do planowania trasy lotu. Biblioteka napisana jest z myślą o technologiach webowych, aplikacja kliencka została więc zrealizowana jako aplikacja webowa. Wykorzystanym w aplikacji frameworkiem `JavaScript` jest `VueJS`.

Rys. 3.6: Widok aplikacji klienckiej, odpowiedzialny za planowanie tras.

3.5. Struktura repozytoriów

3.5.1. Konfiguracja *CI/CD*

i Jak zaznaczono w celu pracy (1.2), system zaopatrzony został w mechanizmy automatyzujące testowanie i wdrażanie nowych funkcjonalności. Popularne narzędzia do ciągłej integracji, takie jak *Jenkins*, *Gitlab CI* czy *Travis CI*, wykorzystują specjalny plik konfiguracyjny umieszczony w repozytorium. Konfiguracja zawiera zestaw kroków, dzięki którym kod obecny w repozytorium zostanie automatycznie zbudowany, przetestowany i wdrożony. Wykorzystywanym w projekcie narzędziem *CI/CD* jest *Gitlab CI*, ktory oczekuje, że w głównym folderze repozytorium będzie znajdował się specjalny plik konfiguracyjny, o nazwie `gitlab-ci.yml`. Jest on obecny we wszystkich repozytoriach projektowych.

3.5.2. Wspólne punkty stykowe – `git submodules`

Komunikacja pomiędzy komponentami systemu realizowana jest za pośrednictwem pakietów `Protobuf`, opisanych w rozdziale 3.2.1. Każde z repozytoriów musi więc posiadać pliki, zawierające definicje pakietów, wygenerowane za pomocą kompilatora `Protobuf`.

Możliwe jest utrzymywanie kopii definicji pakietów w każdym projektowym repozytorium, jednak takie rozwiązanie szybko doprowadzi do konieczności wykonywania dużej ilości ręcznych poprawek w kodzie, za każdym razem gdy zmieni się standard pakietów. Nawet przy niewielkiej liczbie repozytoriów, łatwo tu o błąd programisty.

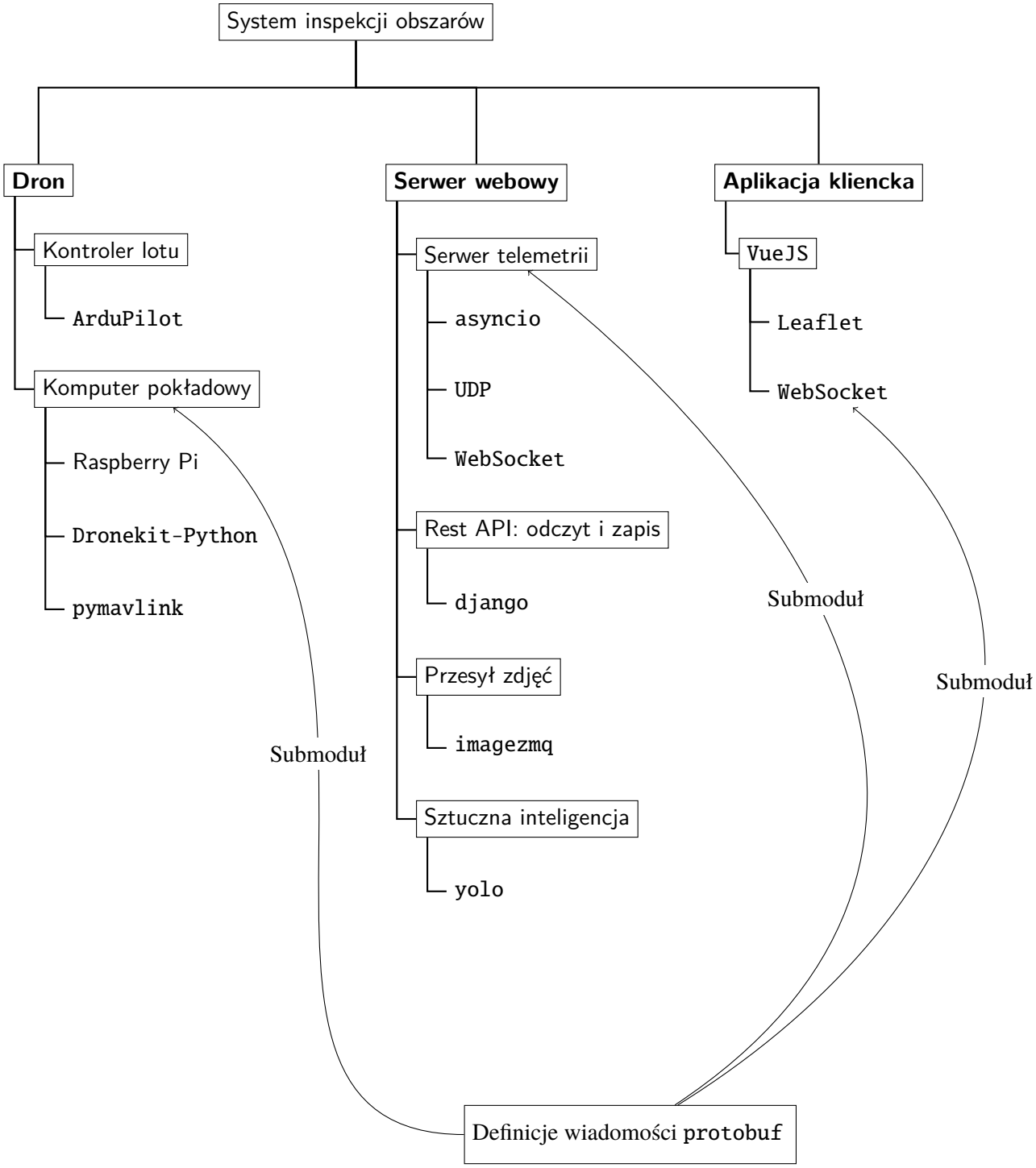
Napisanie biblioteki z definicjami pakietów nie jest tutaj odpowiednim rozwiązaniem, ponieważ biblioteki pisane są zazwyczaj pod konkretny język programowania. Komponenty systemu, które przetwarzają telemetrię są napisane w językach *Python* i *TypeScript* – nie jest więc możliwe napisanie dla nich wspólnej biblioteki.

System kontroli wersji `git` zawiera funkcjonalność `git submodules`[19], która została wykorzystana w projekcie do rozwiązania problemu wspólnych definicji pakietów.

Definicje pakietów, wraz z wygenerowanymi implementacjami w wielu językach, utrzymywane są w osobnym repozytorium, które pobierane jest jako submoduł dla pozostałych projektów. Dzięki temu, możliwa jest aktualizacja repozytorium z definicjami pakietów, a następnie pobranie nowych definicji do wszystkich innych repozytoriów, które wykorzystują submoduł z definicjami.

3.6. Architektura systemu – podsumowanie

Rysunek 3.7 opisuje wykorzystywane w projekcie technologie i sprzęt.



Rys. 3.7: Architektura systemu – wykorzystane technologie

Rozdział 4

Wdrażanie systemu

Jak zaznaczono w zakresie pracy (1.2), architektura systemu ma pozwalać na automatyzację wdrożeń. W niniejszym rozdziale opisane są technologie i praktyki, które zostały zastosowane, aby zautomatyzować wdrażanie systemu.

4.1. Konteneryzacja

Wszystkie elementy systemu (za wyjątkiem oprogramowania na dronie) są uruchamiane w kontenerach. Wykorzystanym systemem konteneryzacji jest `docker`[20]. Konteneryzacja pozwala na zupełne zautomatyzowanie procesu budowania oprogramowania i instalacji koniecznych składników środowiska uruchomieniowego. Aby zbudować kontener zawierający aplikację, programista musi napisać skrypt, w wyniku którego w kontenerze zostanie zainstalowane konieczne do uruchomienia aplikacji oprogramowanie. Proces instalacji i konfiguracji wymaganych bibliotek jest więc automatyczny i deterministyczny – wszystko zależy tylko i wyłącznie od skryptu budującego kontener.

Konteneryzacja ułatwia proces przenoszenia oprogramowania z maszyny na maszynę. Raz zbudowany kontener może zostać uruchomiony na dowolnej nowej maszynie, bez konieczności instalowania czy konfigurowania bibliotek czy programów.

4.1.1. Zachowywanie stanu systemu plików w kontenerze

Ważną różnicą, jaka odróżnia kontenery od aplikacji zainstalowanych w sposób konwencjonalny, jest **bezstanowość**. Jeśli komputer, na którym działają kontenery zostanie zresetowany, bądź kontener zostanie usunięty i utworzony na nowo (na przykład w celu aktualizacji), system plików w kontenerze zostanie zresetowany do stanu, w jakim znajdował się w momencie uruchomienia (takim, jaki jest zapisany w *obrazie kontenera*).

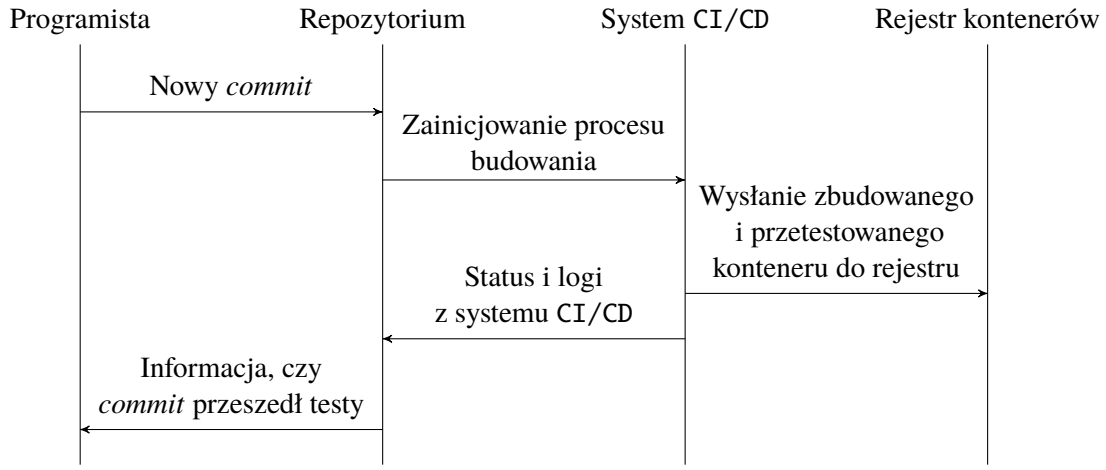
`Docker` pozwala rozwiązać ten problem za pomocą mechanizmu woluminów (*docker volumes*)[21]. Za pomocą woluminów, dowolny fragment systemu plików kontenera może zostać zmapowany do nieulotnej pamięci komputera, na którym działa kontener. W ten sposób zapewnia się persystencję baz danych lub innych plików, które mają być przechowywane przez

aplikację.

Uwaga: nie wszystkie kontenery potrzebują zachowywać pliki – kontener odpowiedzialny za hostowanie statycznej strony internetowej nie potrzebuje zapamiętywać zmian, ponieważ pliki strony nie zmieniają się podczas hostowania.

4.2. Automatyczne budowanie i testowanie komponentów systemu

Jak wspomniano w rozdziale 3.5, poświęconym strukturze repozytoriów, wszystkie repozytoria zawierają plik `.gitlab-ci.yml`. Plik ten definiuje skrypty, jakie wykona system `CI/CD`, gdy do repozytorium zostanie dodany nowy kod. Diagram ilustrujący proces pracy programisty z systemem *CI/CD* zawarty jest na rysunku 4.1.



Rys. 4.1: Diagram sekwencji, opisujący proces automatycznej budowy i testowania kontenerów

Ponieważ konteneryzacja pozwala na zupełne zautomatyzowanie procesu budowania, skrypty w systemie `CI/CD` są w stanie samodzielnie zbudować kontener, zawierający konkretny komponent systemu. Następnie, kontener poddawany jest testom (opisanym w rozdziale 5). Po przejściu testów, kontener wysyłany jest do *rejestru kontenerów* – specjalnego repozytorium, w którym można przechowywać i wersjonować kontenery. Skrypt systemu `CI/CD` automatycznie oznacza kontener za pomocą nazwy brancha, na której został zbudowany. Kontenery pochodzące z brancha *master*, uruchamiane są na serwerze produkcyjnym. Przykładowa konfiguracja *Gitlab CI*, wykorzystywana w repozytorium aplikacji klienckiej opisana została w listingu 4.1.

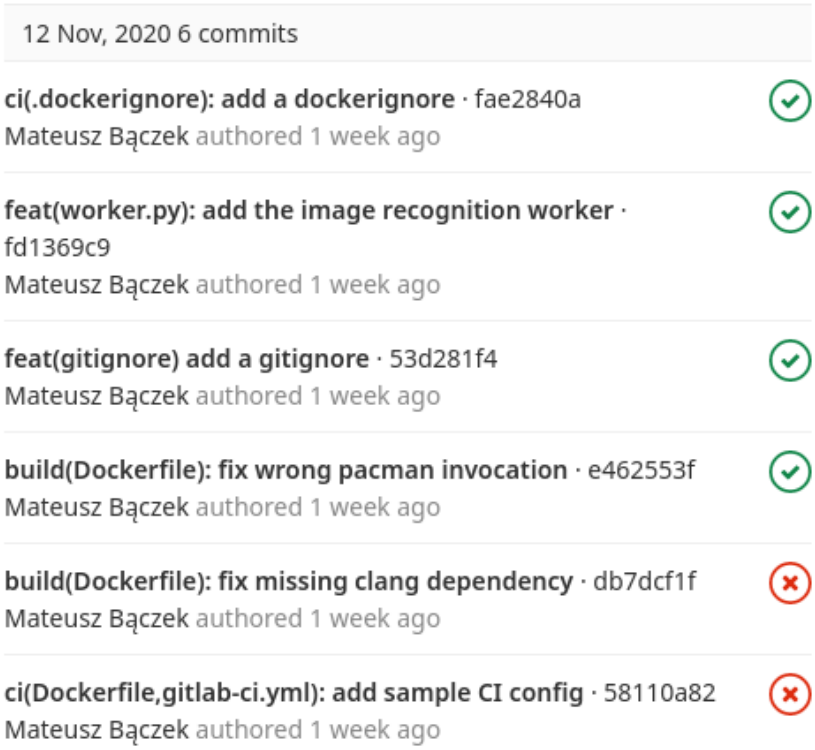
Listing 4.1: Plik konfiguracyjny *Gitlab CI*, budujący aplikację kliencką

```
1 # Definicje zmiennych, wpływających na proces budowania
2 variables:
3   # Wywołuje automatyczne pobranie submodułów,
4   # przed przystąpieniem do budowania
```

```
5   GIT_SUBMODULE_STRATEGY: recursive
6
7   # Zapewnia dostęp do dockera z poziomu systemu CI/CD
8   DOCKER_HOST: tcp://docker:2375
9   DOCKER_TLS_CERTDIR: ""
10
11  # Specyfikuje nazwę kontenera, który zostanie zbudowany
12  IMAGE_NAME: "registry.gitlab.com/academic-aviation-club/gavron/frontend"
13
14 # Przed wykonaniem danego skryptu budującego, wykonywane jest logowanie
15 # do rejestru dockera - umożliwi to zapisanie w nim
16 # zbudowanego obrazu aplikacji. Zmienna $GITLAB_DEPLOY_TOKEN
17 # przypisywana jest w ustawieniach repozytorium
18 before_script:
19   - docker login -u baczek-vps -p $GITLAB_DEPLOY_TOKEN registry.gitlab.com
20
21 # Zawsze budowany jest obraz testowy, który zostanie
22 # później wykorzystany do testów integracyjnych
23 build_testing_image:
24   stage: build
25   # Zapewnia dostęp do dockera w ramach zadania ‘build_testing_image‘
26   services:
27     - docker:19.03.12-dind
28   image: docker:19.03.12
29
30   # Skrypt wpierw buduje obraz dockera, zawierający aplikację.
31   # Następnie uruchamia na obrazie testy jednostkowe. Jeśli testy
32   # nie zwrócą błędu, obraz jest wysyłany do rejestru.
33   # Na obrazach w rejestrze wykonywane są testy integracyjne.
34   script:
35   - docker build -f docker/test.dockerfile -t $IMAGE_NAME:
       ↪ test_$CI_COMMIT_REF_NAME .
36   - docker run $IMAGE_NAME:$CI_COMMIT_REF_NAME "npm run test"
37   - docker push $IMAGE_NAME:test_$CI_COMMIT_REF_NAME
38
39 # Tylko dla brancha ‘master‘, budowany jest obraz
40 # produkcyjny, który zostanie uruchomiony na serwerze
41 build_prod_image:
42   stage: build
43   services:
44   - docker:19.03.12-dind
45   # ‘only‘ pozwala na wybranie brancha, na którym działa dany skrypt
46   only:
47     - master
48   image: docker:19.03.12
49
50   script:
51   - docker build -f docker/Dockerfile -t $IMAGE_NAME:$CI_COMMIT_REF_NAME .
```

```
52   - docker run $IMAGE_NAME:$CI_COMMIT_REF_NAME "npm run test"
53   - docker push $IMAGE_NAME:$CI_COMMIT_REF_NAME
```

Wszystkie logi, jakie system CI/CD zbierze w trakcie budowania i testowania obrazu z aplikacją są dostępne dla programisty – pozwala to odnaleźć błędy, które pojawiły się w procesie budowania oraz sprawdzić wyniki testów. Systemy obsługi repozytorium (taki jak *GitHub*, *BitBucket* czy *GitLab*), wyświetlają informacje o tym, czy dany commit przeszedł testy w systemie *CI/CD*, ułatwiając odnajdywanie commitów zawierających błędy. Przykład tej funkcji zilustrowany jest na rysunku 4.2.



Rys. 4.2: Historia zmian w kodzie w serwisie *GitLab*. Ikony po prawej stronie informują, czy dany commit przeszedł testy w systemie CI/CD

4.3. Automatyczne aktualizacje kontenerów

Pobranie i uruchomienie kontenera, zawierającego dany komponent systemu nie wymaga przechodzenia przez proces instalacji czy konfiguracji – wystarczy pobrać obraz kontenera z rejestru i uruchomić go:

Listing 4.2: Pobranie i uruchomienie obrazu dockera, zawierającego aplikację

```
1 # Zapisane jako zmienna, aby poprawić czytelność przykładu
2 IMAGE=registry.gitlab.com/academic-aviation-club/gavron/frontend:master
3
4 docker pull $IMAGE
5 docker run \
6     -d \ # detach: kontener działa w tle, nie wysyła logów do terminala
```

```
7     -p 5000:5000 \ # mapuje port 5000 z kontenera do portu na komputerze
8     --name frontend \ # Nadaje nazwę uruchomionemu kontenerowi
9     $IMAGE # Nazwa obrazu do uruchomienia
```

W przypadku, gdy na serwerze działa już kontener z aplikacją, należy pobrać nowy obraz kontenera, usunąc obecnie działający kontener i uruchomić nowy obraz:

Listing 4.3: Aktualizacja kontenerów

```
1  IMAGE=registry.gitlab.com/academic-aviation-club/gavron/frontend:master
2
3  docker pull $IMAGE # Pobranie najnowszej wersji kontenera
4
5  # Nazwa kontenera nadana w poprzednim przykładzie, za pomocą --name
6  docker stop frontend
7
8  # Usunięcie kontenera w starej wersji (jego stanu, zmian w systemie plików)
9  docker rm frontend
10
11 # Tak samo jak w przykładzie powyżej
12 docker run \
13     -d \ # detach: kontener działa w tle, nie wysyła logów do terminala
14     -p 5000:5000 \ # mapuje port 5000 z kontenera do portu na komputerze
15     --name frontend \ # Nadaje nazwę uruchomionemu kontenerowi
16     $CONTAINER # Nazwa kontenera do uruchomienia
```

W przypadku wielu równolegle działających kontenerów, ręczna aktualizacja każdej działającej aplikacji jest zadaniem niepotrzebnie czasochłonnym. Wprowadza też dodatkowy punkt, w którym może zajść pomyłka – na przykład pominięcie jednego z kontenerów przy aktualizacji.

Projekt wykorzystuje narzędzie `Ouroboros` do automatycznej aktualizacji uruchomionych na serwerze kontenerów. `Ouroboros` w regularnych odstępach czasu sprawdza, czy uruchomione na serwerze kontenery nie wymagają aktualizacji. Jeśli w rejestrze dostępna jest nowa wersja obrazu kontenera, pobiera ją i zastępuje nią obecnie działający kontener.

Dzięki zastosowaniu narzędzia `Ouroboros`, w trakcie testów w terenie udało się znacznie usprawnić aktualizacje infrastruktury systemu. W przypadku pojawienia się błędu, programiści musieli jedynie zaktualizować kod w repozytorium – system *CI/CD* automatycznie budował poprawioną wersję kontenera, `Ouroboros` aktualizował serwer produkcyjny. Po dodaniu nowego kodu do repozytorium, uaktualniona wersja aplikacji pojawiała się na serwerze po mniej niż pięciu minutach.

4.4. Orkiestracja systemu złożonego z wielu kontenerów – `docker-compose`

Finalnie, infrastruktura internetowa systemu składa się z sześciu jednocześnie działających kontenerów dockerowych:

1. Kontener serwujący stronę internetową z aplikacją kliencką,
2. Kontener z serwerem API,
3. Kontener z serwerem `imagezmq`,
4. Kontener z systemem telemetrii,
5. Kontener z programem `ouroboros` (opisanym w rozdziale 4.3),
6. Kontener z systemem do rozpoznawania obrazów.

Każdy kontener musi zostać odpowiednio skonfigurowany w momencie uruchamiania. Konfiguracja poszczególnego kontenera obejmuje:

- Przypisanie limitów zasobów (n.p. maksymalny procent wykorzystania procesora)
- Udostępnienie portów serwera, które może wykorzystywać dany kontener,
- Przypisanie woluminów systemu plików serwera, w celu zapewnienia persystencji danych, zapisywanych przez kontener (opisane w rozdziale 4.1.1),
- Zdefiniowanie reguł automatycznego restartu kontenera (na przykład gdyby proces działający w kontenerze uległ awarii),
- Określenie zmiennych środowiskowych, które mogą wpływać na zachowanie się procesów wewnątrz kontenera (przykładowo, kontener `ouroboros` pozwala za pomocą zmiennych środowiskowych określić, jak często mają być sprawdzane aktualizacje kontenerów).

Narzędzie `docker-compose` pozwala zebrać całą konfigurację kontenerów do jednego pliku konfiguracyjnego [22]. W ten sposób, po uzyskaniu dostępu do rejestru z kontenerami, zawierającymi komponenty systemu, możliwe jest natychmiastowe pobranie i uruchomienie systemu z właściwą konfiguracją kontenerów.

Rozdział 5

Testy systemu

Jak zaznaczono w celu pracy (1.2), prototyp systemu ma być w pełni testowalny. Testy wykonywane są wewnątrz systemu ciągłej integracji (*CI/CD*), opisanym w rozdziale 4. Poniższy rozdział opisuje rozwiązania zastosowane w celu przetestowania działania systemu, zanim został on uruchomiony na prawdziwym dronie.

5.1. Testy jednostkowe

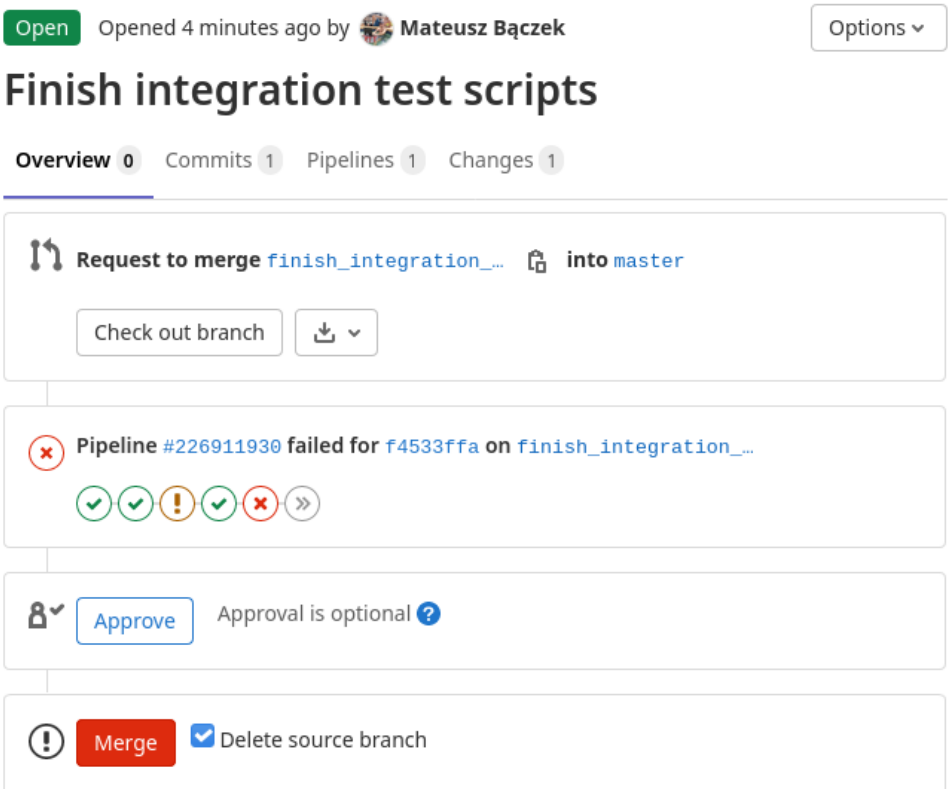
Komponenty systemu zaopatrzone są w testy jednostkowe, co pozwala na sprawdzenie działania kodu w zakresie poszczególnego komponentu. Przykład testu jednostkowego wykorzystywanego w systemie zawarty jest w listingu 5.1.

Listing 5.1: Test jednostkowy modułu sterującego uruchamianiem symulatoru drona. test sprawdza, czy symulator uruchamia się, oraz czy zostaje poprawnie zamknięty po wywołaniu metody `.stop()`.

```
def test_container_lifecycle():
  # Check how many docker containers are running
  num_of_containers = int(check_output('docker ps | wc -l', shell=True))

  print('setting up the runnner')
  runner = SitlDockerHelper('ArduCopter', run_in_background=True)

  print('running the container')
  runner.run()
  sleep(10)

  new_num_of_containers = int(
        check_output('docker ps | wc -l', shell=True)
    )

  # Check if the number of running containers has increased
  assert num_of_containers + 1 == new_num_of_containers

  print('stopping')
```

```
20  runner.stop()
21
22  # Check if the num of running containers is back to the start state
23  assert int(check_output('docker ps | wc -l', shell=True)) ==
      ↪ num_of_containers
```

Wykonywanie testów w systemie *CI/CD* zwiększa sprawność pracy z systemami kontroli wersji. W przypadku implementowania nowej funkcji na gałęzi (*branch*), system *CI/CD* pozwala określić, czy nowa funkcjonalność nie destabilizuje systemu – ostrzegając przed przyjęciem na główną gałąź (*master branch*) kodu, który nie przeszedł testów jednostkowych. Rysunek 5.1 przedstawia przykład takiego zachowania w usłudze *GitLab*.



Rys. 5.1: Widok interfejsu scalania gałęzi w serwisie *GitLab*. Gałąź `finish_integration_tests` nie przechodzi testów w systemie *CI/CD*. Administrator repozytorium natychmiast otrzymuje informację, że kod nie jest jeszcze gotowy do scalenia.

5.2. Testy integracyjne

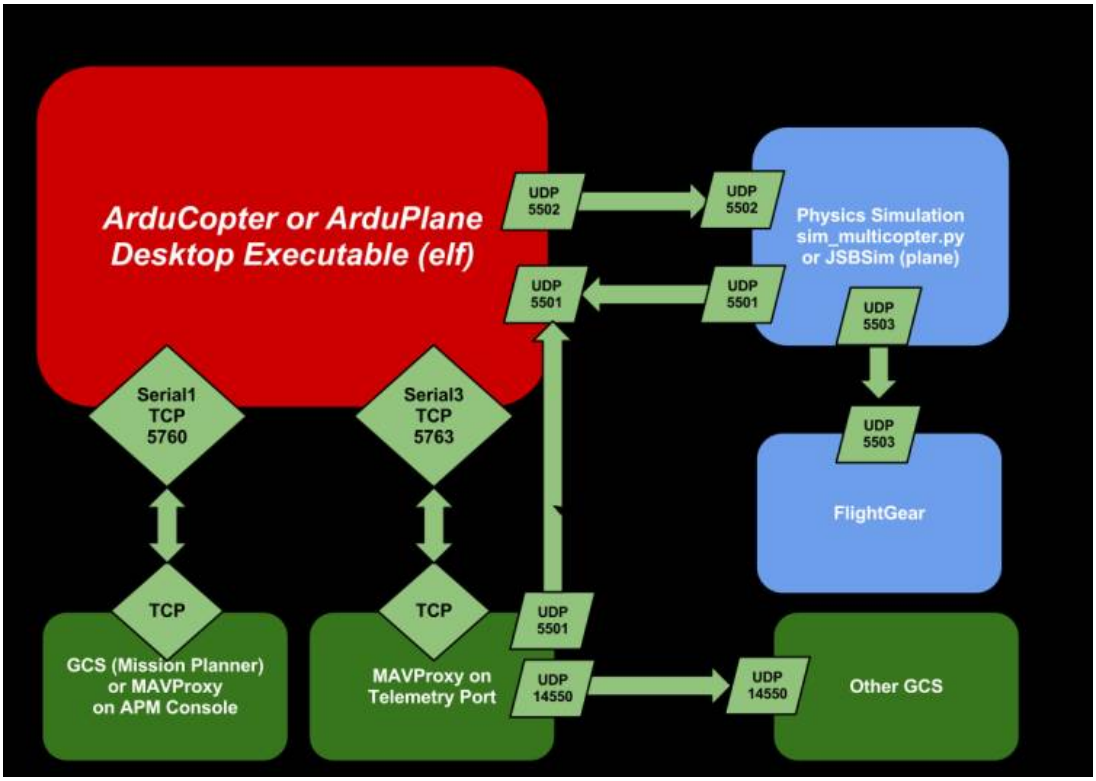
Realizowany w pracy system nie ogranicza się do pojedynczego, samodzielnego komponentu, zawiera kilka równolegle działających usług. testy jednostkowe nie są w takim przypadku wystarczające, aby gruntownie sprawdzić poprawność działania kodu.

Dodatkowym wyzwaniem przy projektowaniu testów jest interakcja z dronem, którego kontroler lotu komunikuje się z systemem za pośrednictwem interfejsu `UART`, podłączonego do komputera pokładowego. Błąd w komunikacji z kontrolerem lotu może doprowadzić do rozbicia ma-

szyny, straty są tutaj zupełnie realne, inaczej niż w przypadku testowania systemów pracujących jedynie na serwerach.

5.2.1. Symulacja i symulatory

Symulatory lotu są szeroko wykorzystywane w przemyśle lotniczym [24]. Amatorskie otwartoźródłowe kontrolery lotu nie są tutaj wyjątkiem – jak wspomniano w rozdziale poświęconym kontrolerom lotu (3.1.1), wykorzystywane w projekcie oprogramowanie *ArduPilot* ma możliwość pracy w trybie *SITL* (*software in the loop*). Jest to możliwe dzięki specjalnym opcjom, które można ustawić w trakcie kompilacji. Opcje umożliwiają przemapowanie interfejsów kontrolujących peryferia samolotu na ich wirtualne odpowiedniki, które podłączane są do symulatora. Kod sterujący zachowaniem drona oraz dane telemetryczne pozostają takie same jak w przypadku pracy na prawdziwej maszynie. Szczegółowy diagram ilustrujący przemapowywane peryferia kontrolera lotu zawarty jest na rysunku 5.2.



Rys. 5.2: Diagram zaczerpięty z dokumentacji *ArduPilota* [11], ilustrujący działanie symulatora SITL.

Ponieważ jedyna zmiana w wynikowym pliku wykonywalnym z kontrolerem lotu to przemapowane interfejsy peryferiów, komunikacja z kontrolerem lotu nadal działa w dokładnie taki sam sposób. Kod, który był w stanie skomunikować się z kontrolerem lotu działającym w trybie *SITL*, będzie na pewno działał również na prawdziwej maszynie (wyjątkiem byłaby oczywiście usterka sprzętowa, jednak takie problemy leżą poza zakresem odpowiedzialności kodu).

Listing 5.2 ilustruje, jak zależnie od parametru konfiguracyjnego, skrypt sterujący dronem będzie próbował podłączyć się do prawdziwej maszyny lub do symulatora. Jedyną różnicą są

parametry w konstruktorze obiektu połączenia, pozostały kod nie musi być modyfikowany gdy przechodzi się z symulatora na prawdziwego drona.

Listing 5.2: Przykład kodu łączącego się z prawdziwym dronem lub z symulatorem lotu, zależnie od wartości parametru `USE_SITL`. Utworzony obiekt `connection`, reprezentujący połączenie, zawsze zachowuje się w taki sam sposób, niezależnie czy wykorzystywana jest prawdziwa maszyna czy symulator.

```
if USE_SITL:
    connection = mavutil.mavlink_connection(
        'udpin:0.0.0.0:14551' # Połączenie z symulatorem na porcie UDP
    )

else:
    connection = mavutil.mavlink_connection(
        '/dev/ttyS0', # Połączenie z dronem za pomocą interfejsu UART
        baud=57600
    )
# Dalszy kod wykorzystujący obiekt ‘connection‘.
# Nie ma znaczenia czy ‘connection‘ oznacza połączenie
# do symulatora czy do prawdziwego drona/samolotu
```

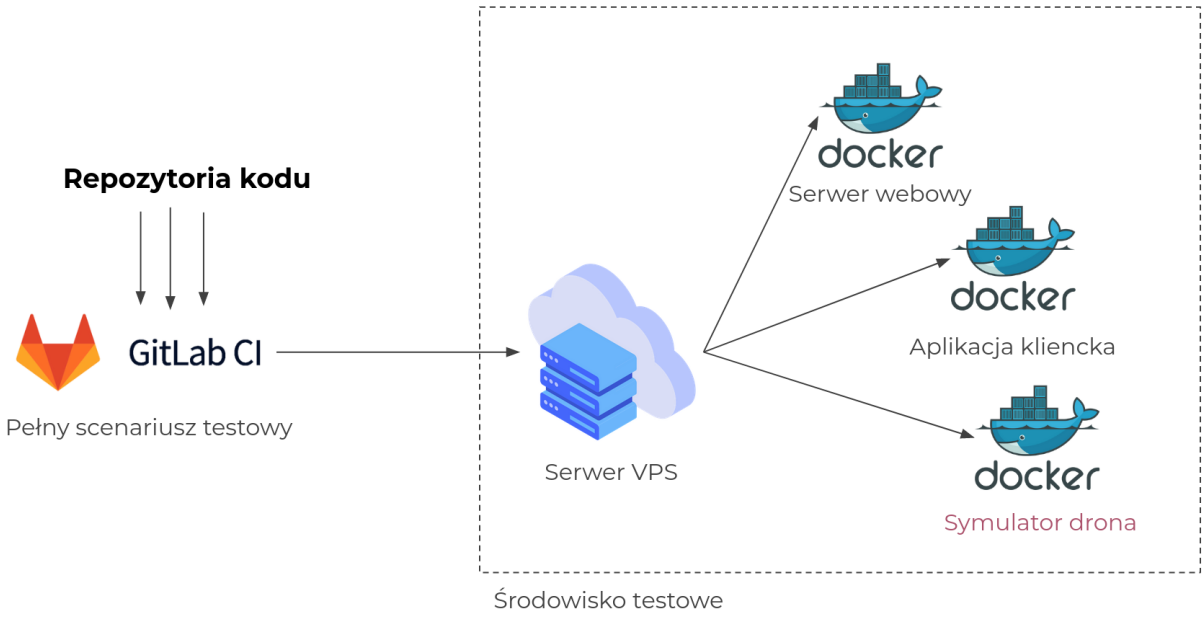
5.2.2. *SITL* i automatyczne testy w systemie *CI/CD*

Jak opisano w rozdziale 4, wszystkie komponenty systemu docelowo uruchamiane są w kontenerach. Takie rozwiązanie pozwala na zagwarantowanie poprawnego wdrożenia systemu na serwerze produkcyjnym. Równocześnie, kontenery wysyłane na serwer produkcyjny można wcześniej przetestować w wirtualnym środowisku testowym.

Na potrzeby projektu, symulator kontrolera lotu został zkonteneryzowany i wykorzystany w *CI/CD* do testów integracyjnych, angażujących wszystkie komponenty systemu. Zespół pracujący nad projektem używał gotowego kontenera z symulatorem, aby przeprowadzać testy na własnych maszynach. Takie rozwiązanie eliminuje wszystkie problemy, jakie mogą pojawić się podczas wykorzystywania oprogramowania którego konfiguracja jest bardzo złożona – tak jak w przypadku symulatora lotu.

5.2.3. Budowanie testów integracyjnych z wykorzystaniem *Gitlab CI*

Do uruchamiania testów integracyjnych przeznaczony został dedytkowany serwer Linux. Serwer służył za platformę, na której uruchamiane były wszystkie komponenty systemu. Diagram 5.3 ilustruje sposób wykorzystywania dedykowanego serwera w celu przeprowadzenia testów integracyjnych.



Rys. 5.3: Pełne testy integracyjne na dedykowanym serwerze.

5.2.4. Konfiguracja serwera zarządzanego przez *Gitlab CI*

Platforma *GitLab* umożliwia uruchamianie testów na własnej infrastrukturze, w ramach usług dostępnych dla darmowych kont. Infrastruktura platformy domyślnie pozwala jedynie na testowanie wewnątrz izolowanych kontenerów `docker`. Jest to rozwiązanie zupełnie wystarczające w przypadku testów jednostkowych. Uruchomienie pełnego systemu w sztucznym środowisku wymaga jednak większej kontroli nad serwerem testowym. *Gitlab CI* pozwala na podłączenie własnych serwerów do swojej sieci – dzięki takiemu rozwiązaniu użytkownicy mogą wykorzystywać pełne możliwości systemu ciągłej integracji.

Podłączony do systemu serwer wykonuje skrypty uruchamiające testy integracyjne bezpośrednio na serwerze, co pozwala na równoległe uruchomienie wielu kontenerów `docker` oraz monitorowanie ich pracy z zewnątrz – jak pokazano na diagramie 5.3.

5.2.5. Definiowanie wieloetapowych testów integracyjnych

W ramach testów integracyjnych testowane są wpierw pojedyncze kontenery – nie są to jednak testy jednostkowe. Uruchamiany jest cały kontener zawierający komponent systemu, następnie jest on testowany przez zewnętrzny skrypt. W przypadku aplikacji klienckiej, uruchamiana jest sterowana automatycznie przeglądarka internetowa, która symuluje zachowanie prawdziwego użytkownika, tworząc nową trasę przelotu drona. W przypadku serwera *REST API*, wykonywane są zapytania *http* weryfikujące poprawność działania *API*.

Po przetestowaniu pojedynych kontenerów, uruchamiane są testy angażujące wiele kontenerów – sprawdzające czy komponenty systemu będą ze sobą współpracowały. Przykładowy scenariusz testowy zawiera:

1. Uruchomienie kontenerów:

- symulatora kontrolera lotu,
- aplikacji sterującej dronem,
- aplikacji klienckiej
- serwera telemetrii
- serwera API.

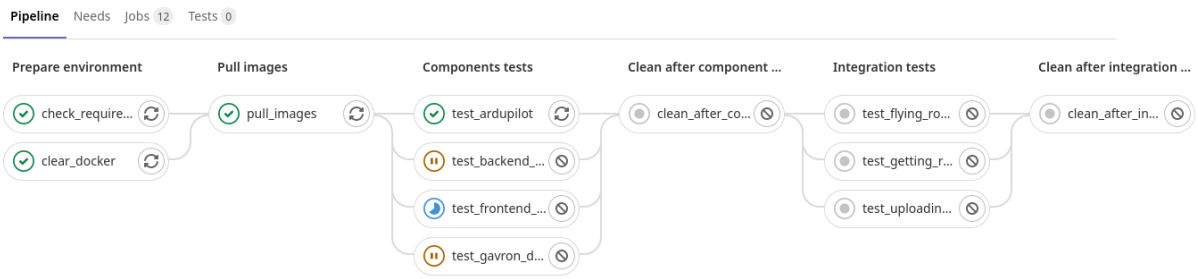
2. Wykorzystanie zdalnie sterowanej przeglądarki w celu dodania nowej trasy przelotu w aplikacji klienckiej.
3. Sprawdzenie, czy w aplikacji klienckiej pojawi się telemetria z drona, wykonującego zaplanowaną trasę.

Taki test angażuje wszystkie uruchamiane komponenty. Skrypt wykonujący test musi jedynie sterować aplikacją kliencką i weryfikować, czy pojawiają się w niej dane pochodzące z pozostałych komponentów systemu.

Etapy testów w *GitLab CI*

Platforma *GitLab CI* pozwala na podzielenie testów na etapy. Każdy etap składa się z określonej liczby zadań. Aby system *CI/CD* mógł przejść do kolejnego etapu, muszą zostać wykonane wszystkie zadania z poprzednich etapów. W przypadku testów integracyjnych oznacza to, że wpierw zweryfikowane musi zostać działanie wszystkich kontenerów składających się na system, następnie testowane są interakcje między komponentami. Finalny zestaw testów integracyjnych (*test pipeline*), które były wykonywane na systemie ilustruje rysunek 5.4.

Uwaga: etapy takie jak *Pull images* czy *Clean after component tests* sprawdzają stan serwera przed i po wykonaniu etapu testów. Jest to wymagane, ponieważ testy nie ograniczają się do izolowanych kontenerów dockera, lecz wymagaja komunikowania się z komponentami za pomocą skryptów uruchamianych bezpośrednio na serwerze testowym. Etapy wykonywane pomiędzy testami weryfikują, czy konkretny etap testów zakończył się w odpowiedni sposób; nie pozostawił na serwerze działających procesów lub kontenerów.



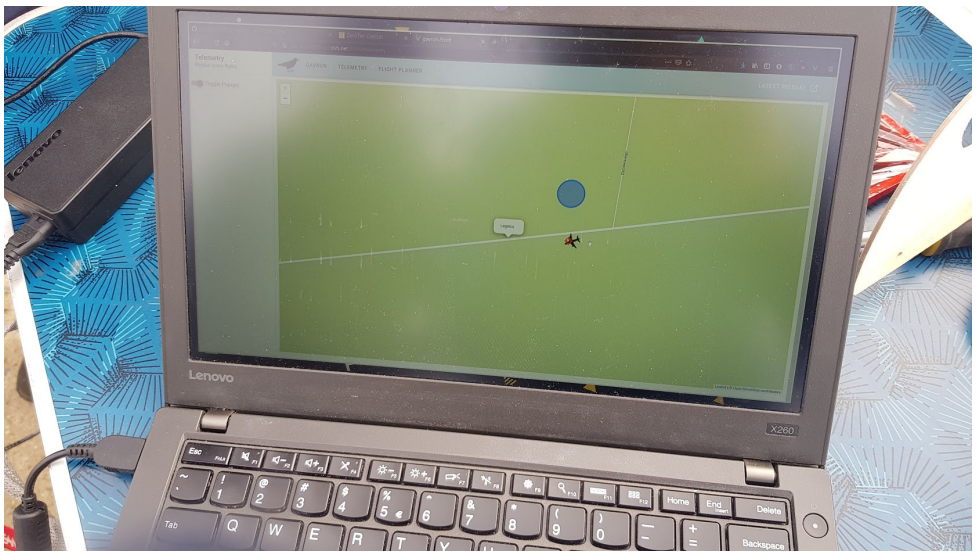
Rys. 5.4: Interfejs *GitLab CI* prezentujący postęp w wykonywaniu wieloetapowych testów integracyjnych.

5.3. Testy w terenie

5.3.1. Wczesne etapy rozwoju systemu

Ze względu na pandemię COVID-19, system testowany był w terenie jedynie trzy razy. Podczas pierwszych testów, dron z kamerą nie był jeszcze przygotowany, system został więc zaadaptowany do wykorzystania w jednym z samolotów Akademickiego Klubu Lotniczego (5.5). Testom została poddana aplikacja kliencka i system obsługi telemetrii.

Rys. 5.5: Pierwsze loty testowe, wykorzystujące jeden z samolotów Akademickiego Klubu Lotniczego. Testy objęły aplikację kliencką i system obsługi telemetrii.



Rys. 5.6: Wczesna wersja aplikacji klienckiej, wyświetlająca pozycję samolotu w trakcie lotu.

Jak wspomniano w podrozdziale 4.3, automatyczne aktualizowanie infrastruktury internetowej systemu pozwoliło na szybsze i bezpieczniejsze wprowadzanie poprawek w systemie. Jest to szczególnie pożądane w przypadku testowania dronów/samolotów, ponieważ przeprowadzenie lotu wymaga obecności całego zespołu: mechaników, elektroników, programistów oraz pilota. Szybsze wprowadzanie zmian w oprogramowaniu umożliwia sprawniejsze wykonywanie lotów i wykonanie większej ilości lotów testowych.

5.3.2. Finalne testy systemu

5.3.3. Platforma testowa

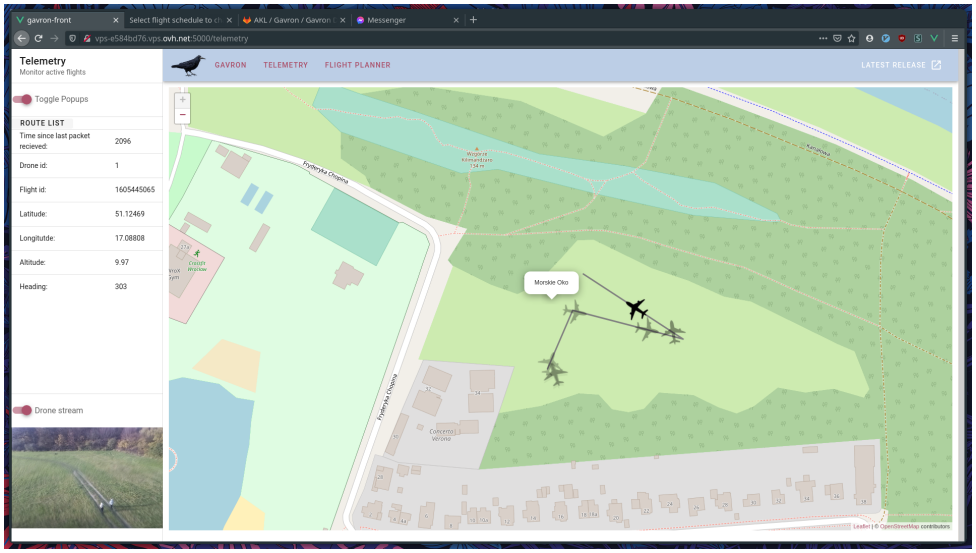
Rys. 5.7: Dron wykorzystywany do finałowych testów projektu.

Finalnie wykorzystywany dron, przedstawiony na rysunku 5.7, zawierał ten sam kontroler lotu co samolot używany do wczesnych testów, opisanych w podrozdziale 5.3.1. Dodatkowo, dron zaopatrzony był w kamerę na gimbalu.

5.3.4. Przebieg lotów testowych

W trakcie lotów testowych kilkakrotnie został wykonany scenariusz wykorzystania systemu, opisany w celu pracy (1.2):

1. W aplikacji klienckiej wyznaczona została nowa trasa lotu,
2. dron pobrał trasę i rozkład lotu z serwera *REST API*,
3. dron wystartował, po czym zaczął wysyłać dane telemetryczne i zdjęcia (5.8),
4. dane telemetryczne i wykonane zdjęcia zostały przekazane do aplikacji klienckiej,
5. odebrane na serwerze zdjęcia zostały przetworzone przez algorytmy sztucznej inteligencji, które rozpoznały obecnych na zdjęciach ludzi (5.9).



Rys. 5.8: Finalna wersja aplikacji klienckiej w trakcie testów w terenie. W prawym dolnym rogu widoczny podgląd zdjęć wykonywanych w trakcie lotu, przesyłanych w czasie rzeczywistym do aplikacji.

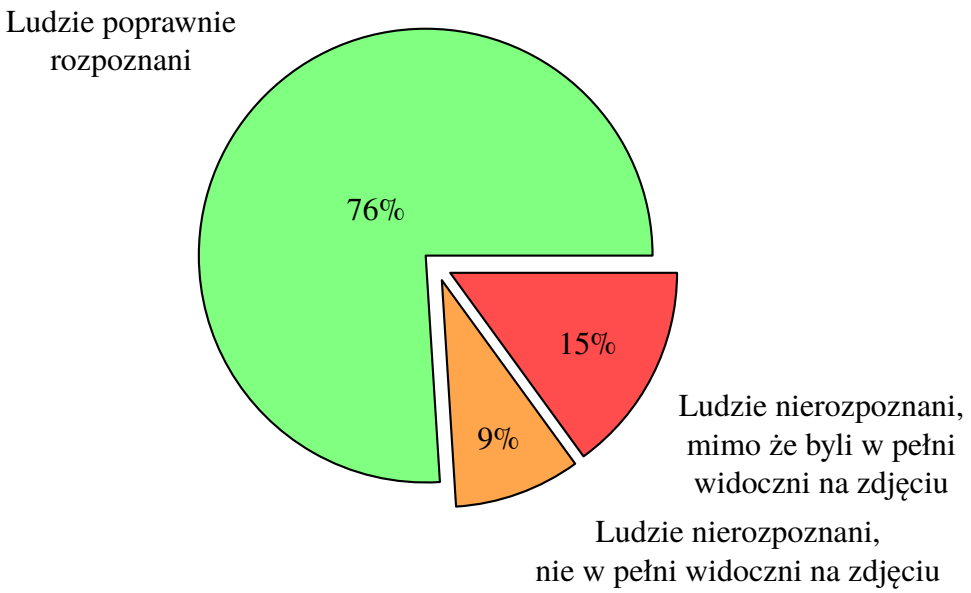
Aplikacja kliencka, widoczna na rysunku 5.8 wyświetla zaplanowaną wcześniej trasę przelotu oraz pozycję drona, odczytaną z danych telemetrycznych. Pozycja maszyny jest aktualizowana w czasie rzeczywistym. Równolegle odbierane są nowe zdjęcia wykonane w czasie lotu.

5.3.5. Rozpoznawanie obiektów na zdjęciach

Rys. 5.9: Zdjęcie wykonane podczas lotu, na zdjęciu poprawnie rozpoznano dwie osoby.

Rys. 5.10: Zdjęcie wykonane podczas lotu. Samochód został niepoprawnie oznaczony jako osoba.

Algorytmy sztucznej inteligencji, omówione w podrozdziale 3.3.3 próbowały rozpoznać ludzi na zdjęciach wykonanych w trakcie lotów. Większość zdjęć została rozpoznana poprawnie, system nie rozpoznawał głównie zdjęć, które były niewyraźne. Rysunki 5.9 i 5.10 zawierają przykładowe zdjęcia wraz z oznaczonymi na obiektami. Diagram 5.11 zawiera podsumowanie dokładności rozpoznawania ludzi, obliczonej na podstawie wszystkich zdjęć wykonanych podczas lotów.



Rys. 5.11: Dokładność z jaką rozpoznawani są ludzie na zdjęciach wykonanych podczas lotów

Rozdział 6

Podsumowanie

W trakcie testów w terenie wszystkie komponenty systemu bezkonfliktowo współgrały ze sobą. Jest to zasługa faktu, że były wcześniej regularnie sprawdzane wewnątrz środowiska testowego. Z punktu widzenia infrastruktury sieciowej, testy w terenie nie różniły się niczym od testów wykonywanych na symulatorze – dane telemetryczne odbierane z drona były dokładnie takie same.

Osiągniętą dokładność rozpoznawania ludzi na zdjęciach wykonanych w trakcie lotu można uznać za zadowalającą, zważywszy na fakt że praca nie skupiała się na algorytmach rozpoznawania obrazu. Wykorzystanie otwartych zbiorów danych, zawierających oznakowane zdjęcia wykonane z samolotów i dronów (przykładowo *VisDrone Dataset* [25]), może poprawnić dokładność rozpoznawania obiektów na zdjęciach.

Jak wspomniano w podrozdziale 5.3.1, automatyczne aktualizowanie infrastruktury internetowej systemu pozwoliło na szybsze i bezpieczniejsze wprowadzanie poprawek w systemie. Dzięki temu udało się uniknąć potencjalnych błędów przy wdrażaniu nowych funkcjonalności.

Zastosowane w pracy rozwiązania kwalifikują się do metodyki *DevOps*, opierającej się na zacieśnieniu więzów pomiędzy programistami i administratorami systemu. Automatyzacja procesów związanych z testami i wdrożeniami oraz powiązanie ich z repozytorium projektowym zwiększa u programistów świadomość tego, jak ważny jest proces wdrażania. W przypadku systemu wykorzystującego realne, fizyczne i drogie komponenty, pozwala to przyspieszyć „dojrzewanie" systemu – czas, po którym programiści mogą być pewni stabilności działania produktu, nad którym pracują.

W początkowej fazie rozwoju projektu, dodatkowy wysiłek związany z budową infrastruktury odpowiedzialnej za wdrożenia może wydawać się zupełnie zbędny. Nie jest to jednak błąd, jak w przypadku przedwczesnej optymalizacji. Wczesne zdefiniowanie ram projektu pozwala na utrzymanie rozwoju kodu w ryzach. Jest niezbędne w celu utrzymania stałego kursu ku wyznaczonym w projekcie celom.

Literatura

[1] UAV Challenge , "Sponsors and supporters 2019 & 2020," 2019. https://uavchallenge.org/about/sponsors-and-supporters/.

[2] F. Remondino, L. Barazzetti, F. Nex, M. Scaioni, and D. Sarazzi, "Uav photogrammetry for mapping and 3d modeling-current status and future perspectives," vol. XXXVIII-1/C22, 01 2011.

[3] Amazon Inc, "Amazon prime air," 2013. https://www.amazon.com/Amazon-Prime-Air/b?ie=UTF8&node=8037720011.

[4] "Akademicki klub lotniczy - strona internetowa," Dostęp z 2020. https://akl.pwr.edu.pl/.

[5] "Wrocławski fundusz aktywności studenckich - strona internetowa," Dostęp z 2020. https://wca.wroc.pl/fast-fundusz-aktywnosci-studenckiej.

[6] E. S. M. Ebeid, M. Skriver, and J. Jin, "A survey on open-source flight control platforms of unmanned aerial vehicle," 08 2017.

[7] "Ardupilot - strona domowa projektu," Dostęp z 2020. https://ardupilot.org/.

[8] "Autoquad - historia projektu," Dostęp z 2020. http://autoquad.org/home/autoquad-project-timeline/.

[9] "Librepilot - strona domowa projektu," Dostęp z 2020. https://www.librepilot.org/site/index.html.

[10] "Px4 autopilot - strona domowa projektu," Dostęp z 2020. https://px4.io/.

[11] "Ardupilot - dokumentacja funkcjonalności symulatora autopilota," Dostęp z 2020. https://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html.

[12] "Ardupilot - dział dokumentacji poświęcony protokołowi mavlink," Dostęp z 2020. https://ardupilot.org/dev/docs/mavlink-commands.html.

[13] "Dronekit - strona domowa projektu," Dostęp z 2020. https://dronekit.io/.

[14] "Protocol buffers - strona domowa projektu," Dostęp z2020. https://developers.google.com/protocol-buffers.

[15] "Protokół websocket - specyfikacja," Dostęp z 2020. `tools.ietf.org/html/rfc6455`.

[16] "Biblioteka `asyncio` - dokumentacja," Dostęp z 2020. `https://docs.python.org/3/library/asyncio.html`.

[17] R. T. Fielding and R. N. Taylor, "Principled design of the modern web architecture," *ACM Trans. Internet Technol.*, vol. 2, p. 115–150, May 2002.

[18] A. F. Joseph Redmon, "Yolov3: An incremental improvement,"

[19] "Dokumentacja systemu kontroli `git` - rozdział poświęcony submodułom," Dostęp z 2020. `https://git-scm.com/book/en/v2/Git-Tools-Submodules`.

[20] "Dokumentacja platformy `Docker`," Dostęp z 2020. `https://docs.docker.com/`.

[21] "Dokumentacja platformy `Docker` - dział poświęcony woluminom `docker volumes`," Dostęp z 2020. `https://docs.docker.com/storage/volumes/`.

[22] "Dokumentacja platformy `Docker` - dział poświęcony narzędziu `docker compose`," Dostęp z 2020. `https://docs.docker.com/compose/`.

[23] J. Baudrillard, *Simulation and simulacra*. The University of Michigan Press, 1994.

[24] J. Leski, *Symulacja i symulatory*. Wydawnictwo Ministerstwa Obrony Narodowej, 1971.

[25] "Dataset *VisDrone*," Dostęp z 2020. `https://github.com/VisDrone/VisDrone-Dataset`.

Spis rysunków

Spis listingów

Spis tabel