

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

OCENA PRACY:

# Spis treści

|   |           |
|---|-----------|
| <b>Spis rysunków</b>  | <b>4</b>  |
| <b>Spis listingów</b>   | <b>4</b>  |
| <b>Spis tabel</b>   | <b>5</b>  |
| <b>1. Wstęp</b>   | <b>7</b>  |
| 1.1. Geneza pracy   | 7         |
| 1.2. Cel pracy  | 8         |
| 1.3. Zakres pracy   | 8         |
| <b>2. Wymagania funkcjonalne systemu</b>                                | <b>9</b>  |
| 2.1. Oprogramowanie na dronie   | 9         |
| 2.2. Protokoły wymiany danych   | 9         |
| 2.3. Oprogramowanie serwerowe   | 10        |
| 2.4. Oprogramowanie klienckie   | 10        |
| <b>3. Architektura systemu: przegląd i wybór technologii</b>            | <b>11</b> |
| 3.1. Zarys architektury   | 11        |
| 3.2. Dron   | 12        |
| 3.3. Protokoły wymiany danych   | 13        |
| 3.4. Oprogramowanie serwerowe   | 16        |
| 3.5. Oprogramowanie klienckie   | 19        |
| 3.6. Struktura repozytoriów   | 20        |
| 3.7. Architektura systemu - podsumowanie                                | 21        |
| <b>4. Wdrażanie systemu</b>   | <b>22</b> |
| 4.1. Konteneryzacja   | 22        |
| 4.2. Automatyczne budowanie komponentów systemu                         | 23        |
| 4.3. Automatyczne aktualizacje kontenerów                               | 25        |
| 4.4. Orkiestracja systemu złożonego z wielu kontenerów - docker-compose | 26        |
| <b>5. Testy systemu</b>   | <b>28</b> |

|   |           |
|---|-----------|
| 5.1. Testy jednostkowe . . . . .          | 28        |
| 5.2. Testy integracyjne . . . . .         | 28        |
| 5.3. Systemy ciągłej integracji . . . . . | 28        |
| 5.4. Testy w terenie . . . . .            | 28        |
| <b>6. Podsumowanie . . . . .</b>          | <b>29</b> |
| 6.1. Wyniki testów . . . . .              | 29        |
| 6.2. Osiągnięta sprawność . . . . .       | 29        |
| 6.3. Pola do poprawy . . . . .            | 29        |
| 6.4. Wnioski . . . . .                    | 29        |
| <b>Literatura . . . . .</b>               | <b>30</b> |
| <b>Indeks rzeczowy . . . . .</b>          | <b>31</b> |

# Spis rysunków

|   |    |
|---|----|
| 3.1. Zarys architektury systemu . . . . .   | 11 |
| 3.2. Diagram przedstawiający system generowania implementacji protokołu, na podstawie narzędzia <code>protobuf</code> . . . . .   | 14 |
| 3.3. Architektura systemu obsługującego telemetrię . . . . .  | 16 |
| 3.4. Zadania wykonywane równolegle wewnątrz systemu obsługi telemetrii. Wspólny interfejs obsługi korutyn oraz wymianę danych pomiędzy zadaniami zapewnia biblioteka <code>asyncio</code> . . . . . | 17 |
| 3.5. Zdjęcie wykonane w czasie lotów testowych. System rozpoznał na nim 2 osoby. . .  | 18 |
| 3.6. Widok aplikacji klienckiej, odpowiedzialny za planowanie tras. . . . .   | 19 |
| 3.7. Architektura systemu - wykorzystane technologie . . . . .  | 21 |
| 4.1. Historia zmian w kodzie w serwisie <i>GitLab</i> . Ikony po prawej stronie informują, czy dany commit przeszedł testy w systemie CI/CD . . . . .   | 25 |
| 4.2. Diagram sekwencji, opisujący proces automatycznej budowy kontenerów . . . . .  | 25 |

# Spis listingów

|  |    |
|--|----|
| 3.1. Przykład definicji pakietu <code>protobuf</code> . . . . .                    | 14 |
| 3.2. Przykład wykorzystania wygenerowanej implementacji pakietów w języku Python . | 15 |
| 4.1. Plik konfiguracyjny <i>Gitlab CI</i> , budujący aplikację kliencką . . . . .  | 23 |
| 4.2. Pobranie i uruchomienie obrazu dockera, zawierającego aplikację . . . . .     | 25 |
| 4.3. Aktualizacja kontenerów . . . . .   | 26 |

# Spis tabel

|  |    |
|--|----|
| 3.1. Najpopularniejsze otwarte projekty oprogramowania obsługującego kontrolery lotu | 12 |
|--|----|

# Skróty

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

**CI/CD** (ang. *Continuous Integration/Continuous deployment*) - zestaw praktyk i narzędzi, wykorzystywanych w celu automatyzacji procesu testowania 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

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 [1]. Projekt został sfinansowany przez Wrocławski Fundusz Aktywności Studenckiej [2].

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 [3]. 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 zastosowanie w dziedzinach takich jak górnictwo, geodezja czy militaria [4], natomiast dostarczanie towarów indywidualnym klientom z pomocą dronów jest w stanie zrewolucjonizować przemysł transportowy.

Przeprowadzenie autonomicznej misji bezzałogową maszyną możliwe jest dzięki osiągnięciom inżynierii oprogramowania. Entuzjaści programowania regularnie udoskonalają i rozbudowują projekty open-source. Powszechna dostępność i możliwość korzystania ze sprawdzonych i stabilnych rozwiązań zdecydowanie usprawnia proces realizowania kreatywnych i innowacyjnych projektów.

## 1.2. Cel pracy

Celem pracy jest stworzenie prototypu systemu inspekcji terenów, wykorzystującego autonomiczne drony. System ma integrować się z już istniejącym oprogramowaniem sterującym autonomicznymi maszynami, oraz wykorzystywać napisaną na potrzeby pracy infrastrukturę informatyczną, pozwalającą na planowanie tras lotów, obsługę telemetrii i rozpoznawanie obiektów na zdjęciach wykonanych w czasie lotu, za pomocą algorytmów sztucznej inteligencji. Finalnie, system ma generować raport podsumowujący każdy lot, w którym zawarta będzie trasa pokonana przez drona i zdjęcia wykonane w trakcie lotu, wraz z rozpoznanymi na nich obiektami.

Architektura systemu musi pozwalać na zautomatyzowanie procesu wdrażania systemu, oraz zautomatyzowanie wdrażania nowych funkcjonalności - każde z wdrożeń musi być poprzedzone testami integracyjnymi na poziomie całego systemu.

Prototyp ma być w pełni testowalny, zarówno na poziomie pojedynczych elementów systemu jak i na poziomie integracji całego projektu - testy muszą angażować wszystkie komponenty systemu, uruchomione wewnątrz w pełni zautomatyzowanego środowiska testowego.

## 1.3. Zakres pracy

Zakres pracy obejmuje elementy projektu związane z inżynierią i architekturą oprogramowania – proces projektowania struktury systemu, wybór technologii, zaprojektowanie punktów stykowych w systemie, automatyzacja procesu wdrażania systemu i nowych funkcjonalności.

Praca opisuje także sposób testowania systemu – od weryfikacji poprawności działania poszczególnych komponentów, po pełne automatyczne testy integracyjne, wykorzystujące wszystkie komponenty systemu wraz ze zintegrowanym symulatorem drona.



## Rozdział 2

# 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 telemetry

Aby umożliwić diagnozowanie stanu systemu w czasie rzeczywistym – szczególnie stanu wykonujących lot wielowirnikowców, telemetry 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 telemetry i zdjęć z dronów w czasie rzeczywistym,
3. przegląd i analiza telemetry oraz zdjęć zarchiwizowanych z poprzednich lotów.

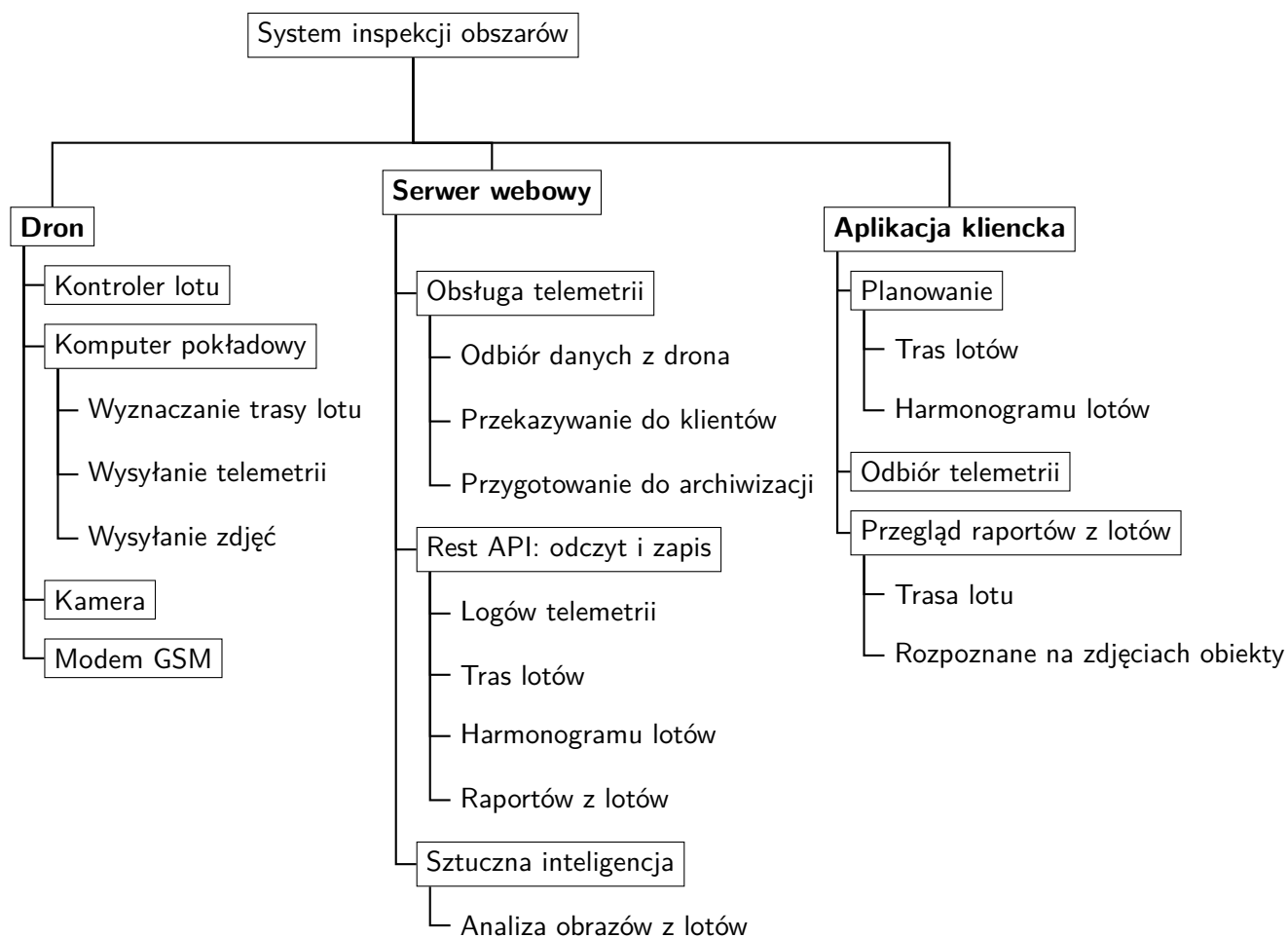
Kluczowym elementem aplikacji klienckiej jest obsługa mapy – wszystkie wymienione funkcjonalności wymagają wizualizacji nadchodzących danych geograficznych, rozszerzonych o dodatkowe informacje (na przykład godzina przelotu przez dany punkt lub zarejestrowane w danym miejscu zdjęcia i wykryte na nich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

## Rozdział 3

# Architektura systemu: przegląd i wybór technologii

### 3.1. Zarys architektury

Rys. 3.1: Zarys architektury systemu



## 3.2. Dron

### 3.2.1. 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 [5] 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[6]     | 2009          | otwarte mikrokontrolery ARM                                       | GPLv3    |
| AutoQuad[7]      | 2011          | mikrokontrolery STM Cortex M4                                     | GPLv3    |
| LibrePilot[8]    | 2015          | zamknięte źródłowo kontrolery lotu, bazujące na architekturze ARM | GPLv3    |
| PX4 Autopilot[9] | 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[10], 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.

### 3.2.2. Komputer pokładowy

Poza kontrolerem lotu, który zawiera jedynie oprogramowanie niezbędne do sterowania dronem i udostępniania strumienia telemetrii, 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 [11].

Infrastruktura projektu ArduPilot dostarcza gotowe narzędzia 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*[12], która rozbudowuje *pymavlink* o w pełni obiektowy, wysokopoziomowy interfejs do komunikacji z kontrolerem lotu. Skrypty odpowiedzialne za logikę biznesową napisane są w Pythonie.

## 3.3. Protokoły wymiany danych

### 3.3.1. 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 telemetrykę z drona napisane jest w Pythonie, interfejs webowy wyświetlający telemetrykę 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 miałyby być dodany kolejny komponent. Jest to potencjalne źródło problemów związanych z integracją między komponentami systemu, ponieważ

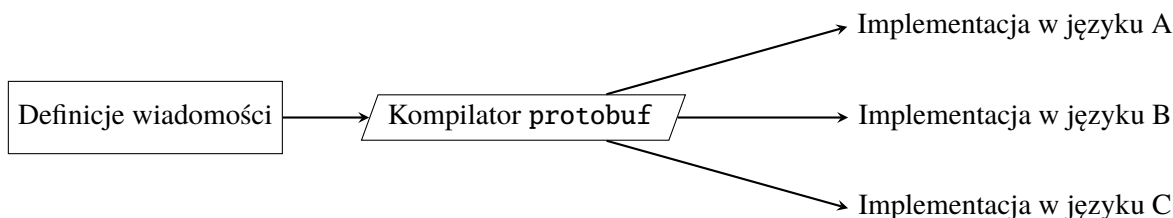
wraz z dodawaniem kolejnych danych telemetrycznych i zwiększaniem liczby wspieranych języków, rośnie szansa na popełnienie błędu w implementacji protokołu. Nawet w przypadku dwóch równolegle wspieranych języków, utrzymywanie spójnej implementacji protokołu nie jest zadaniem trywialnym – szczególnie jeśli 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 takie generują implementację protokołu w wybranych językach programowania. Dzięki automatycznemu generowaniu implementacji protokołu, programiści nie muszą dbać o spójność implementacji pomiędzy wieloma językami.

Wykorzystywanym w projekcie narzędziem do generowania implementacji protokołu jest **protobuf** (*Protocol Buffers*)[13] – 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 upakuje dane w formie binarnej,
- kompatybilność wsteczną – w przypadku dodania nowego typu pakietu telemetry, 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 telemetry 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

```

1 syntax = "proto3";
2
3 package Telemetry;
4
5 /**
6  Wiadomość zawierająca pozycję
7  maszyny w trakcie lotu
8  */
9 message Position {
10     float lat = 1;
11     float lng = 2;
12     float alt = 3;
13     float heading = 4;
14 }
15
16 message TelemFrameHeader {
17     /* Flight metadata */
18     int32 timestamp = 1;
19     int32 machine_id = 2;
20     int32 flight_id = 3;
21
22     /**
23      Kompozycja wcześniej
24      zdefiniowanej wiadomości
25      */
26     Position position = 4;
27 }

```

Listing 3.2: Przykład wykorzystania wygenerowanej implementacji pakietów w języku Python

```

1 # Wygenerowany moduł, zawierający definicje
2 # pakietów, domyślnie nosi nazwę definitions_pb2
3 import definitions_pb2
4 import time
5
6 header = definitions_pb2.TelemFrameHeader()
7
8 header.timestamp = int(time.time())
9 header.machine_id = 1
10 header.flight_id = 5
11
12 header.position.lat = 5
13 header.position.lng = 10
14 header.position.alt = 15
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)

```

### 3.3.2. Zdjęcia wysyłane w trakcie lotu - biblioteka `imagezmq`

Wykonane przez drony zdjęcia wysyłane są na serwer webowy za pomocą biblioteki `imagezmq`. Jak zaznaczono we wstępie, wysyłanie strumienia video 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.

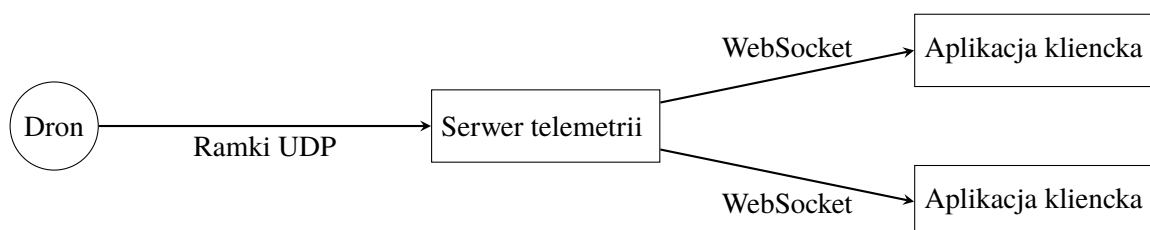
## 3.4. Oprogramowanie serwerowe

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.

### 3.4.1. Obsługa telemetrii

System obsługujący dane telemetryczne odbiera nadchodzące z maszyn pakiety `protobuf`, 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 wszystkie 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 telemetry, 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[14], obsługiwany przez wszystkie powszechnie używane przeglądarki internetowe.

System obsługujący telemetry 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[15]. `Asyncio` pozwala na łatwe zinterfejsowanie ze sobą asynchronicznie działających korutyn, jednocześnie wykonujących różne operacje sieciowe.

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



### 3.4.2. Rest API

System zaopatrzony jest w standardowy serwer Rest API[16], 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).

### 3.4.3. 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* [17]. 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 pojawiają.

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



## 3.5. 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 telemetry w czasie lotu,
- przegląd danych teletycznych zapisanych z poprzednich lotów:
  - podgląd trasy pokonanej przez maszynę,
  - podgląd wykonanych przez maszynę zdjęć, wraz z informacjami o rozpoznanych obiektach,

Najbardziej wymagającym z wymagań jest obsługa mapy, która będzie obecna we wszystkich widokach aplikacji. W systemie została wykorzystana biblioteka Leaflet, służąca do wizualizowania danych na mapach. 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.



## 3.6. Struktura repozytoriów

### 3.6.1. Konfiguracja CI/CD

Jak zaznaczono w zakresie pracy (1.3), system zaopatrzony jest 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*, który oczekuje, że w głównym folderze repozytorium będzie znajdował się specjalny plik konfiguracyjny, o nazwie `gitlab-ci.yml`. Jest on obecny we wszystkich repozytoriach projektowych.

### 3.6.2. Wspólne punkty stykowe - `git submodules`

Komunikacja pomiędzy komponentami systemu realizowana jest za pośrednictwem pakietów *Protobuf*, opisanych w rozdziale 3.3.1. 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.

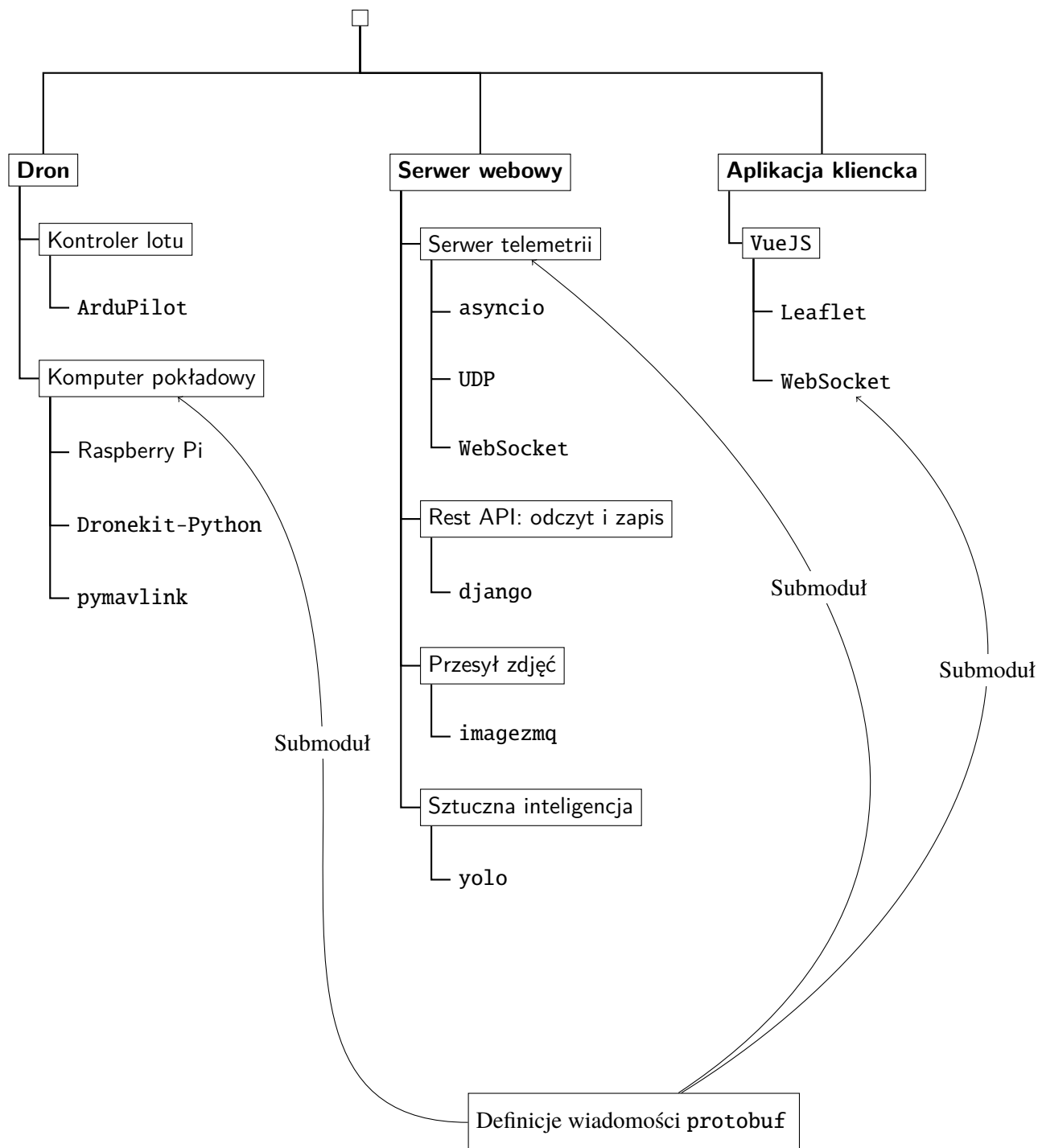
Napisanie biblioteki z definicjami pakietów nie jest tutaj odpowiednim rozwiązaniem, ponieważ biblioteki pisane są zazwyczaj pod konkretny język programowania. Komponenty systemu, które przetwarzają telemetrię są napisane w językach *Python* i *TypeScript* – nie jest więc możliwe napisanie dla nich wspólnej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

System kontroli wersji `git` zawiera funkcjonalność `git submodules`[18], 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.

## 3.7. Architektura systemu - podsumowanie

Rys. 3.7: Architektura systemu - wykorzystane technologie



## Rozdział 4

# Wdrażanie systemu

Jak zaznaczono w zakresie pracy (1.3), architektura systemu ma pozwalać na automatyzację wdrożeń. W poniższym rozdziale opisane są technologie i praktyki, które zostały zastosowane, aby zautomatyzować wdrażanie systemu oraz proces dodawania do systemu nowych funkcjonalności.

### 4.1. Konteneryzacja

Wszystkie elementy systemu docelowo są uruchamiane w kontenerach. Wykorzystanym systemem konteneryzacji jest `docker`[19]. Konteneryzacja ułatwia proces przenoszenia oprogramowania z maszyny na maszynę. Pozwala też na pełne zautomatyzowanie procesu budowania oprogramowania i instalacji koniecznych składników środowiska uruchomieniowego. Konteneryzacja aplikacji wymusza na programiście napisanie skryptu, w wyniku którego w kontenerze zostanie zainstalowane wszystkie konieczne oprogramowanie.

#### 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 zwanego woluminami (*docker volumes*)[20]. 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 plików baz danych, czy innych plików, które mają być przetrzymywane przez aplikację.

*Uwaga:* nie wszystkie kontenery potrzebują zachowywać pliki - kontener odpowiedzialny za hostowanie statycznej strony internetowej nie potrzebuje zapamiętywać zmian w plikach strony.

## 4.2. Automatyczne budowanie komponentów systemu

Jak wspomniano w rozdziale 3.6, 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 (przykładowa konfiguracja *Gitlab CI* dla repozytorium z aplikacją kliencką opisana w listingu 4.1).

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 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.

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 w 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}
```

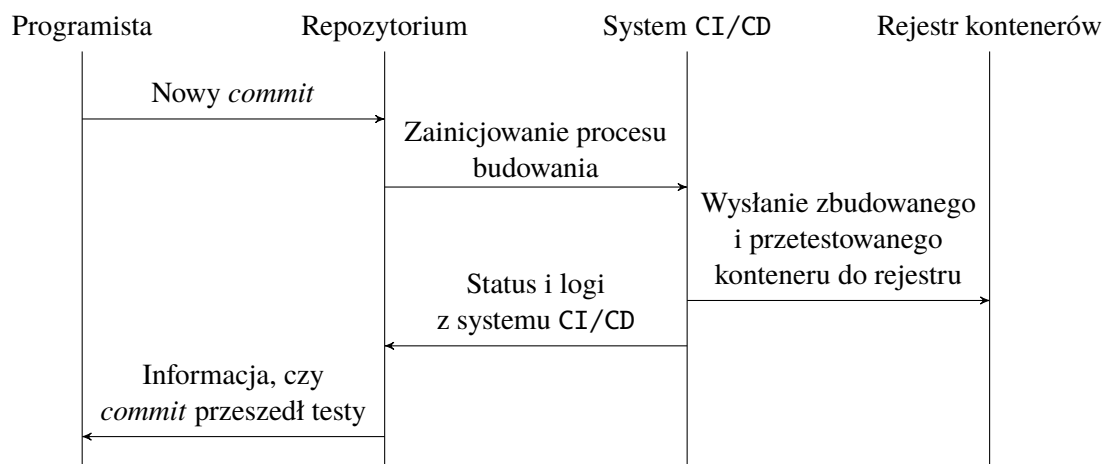
Wszystkie logi, jakie system CI/CD zbierze w trakcie budowania obrazu z aplikacją są dostępne dla programisty - pozwala to odnaleźć błędy, które pojawiły się w procesie budowania. Systemy obsługi repozytorium (taki jak *GitHub*, *BitBucket* czy *GitLab*), wyświetlają informacje o tym, czy dany commit przeszedł testy w systemie CI/CD:



Rys. 4.1: Historia zmian w kodzie w serwisie *GitLab*. Ikony po prawej stronie informują, czy dany commit przeszedł testy w systemie CI/CD

|   |                                    |   |
|---|------------------------------------|---|
| 12 Nov, 2020 6 commits  |                                    |   |
| ci(.dockerignore): add a dockerignore · fae2840a              | Mateusz Bączek authored 1 week ago | ✓ |
| feat(worker.py): add the image recognition worker · fd1369c9  | Mateusz Bączek authored 1 week ago | ✓ |
| feat(gitignore) add a gitignore · 53d281f4                    | Mateusz Bączek authored 1 week ago | ✓ |
| build(Dockerfile): fix wrong pacman invocation · e462553f     | Mateusz Bączek authored 1 week ago | ✓ |
| build(Dockerfile): fix missing clang dependency · db7dcf1f    | Mateusz Bączek authored 1 week ago | ✗ |
| ci(Dockerfile,gitlab-ci.yml): add sample CI config · 58110a82 | Mateusz Bączek authored 1 week ago | ✗ |

Rys. 4.2: Diagram sekwencji, opisujący proces automatycznej budowy kontenerów



## 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, usunąć 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 zająć 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.

## 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 telemetry,
5. Kontener z programem ouroboros,
6. Kontener z systemem do rozpoznawania obrazów.

Każdy kontener musi zostać odpowiednio skonfigurowany w momencie uruchamiania, konfiguracja 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 [21]. 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.

### **5.1. Testy jednostkowe**

### **5.2. Testy integracyjne**

#### **5.2.1. Symulacja i symulatory**

### **5.3. Systemy ciągłej integracji**

### **5.4. Testy w terenie**

## **Rozdział 6**

# **Podsumowanie**

**6.1. Wyniki testów**

**6.2. Osiągnięta sprawność**

**6.3. Pola do poprawy**

**6.4. Wnioski**

# Literatura

- [1] "Akademicki klub lotniczy - strona internetowa," Dostęp z 2020. <https://akl.pwr.edu.pl/>.
- [2] "Wrocławski fundusz aktywności studenckich - strona internetowa," Dostęp z 2020. <https://wca.wroc.pl/fast-fundusz-aktywnosci-studenckiej>.
- [3] UAV Challenge , "Sponsors and supporters 2019 & 2020," 2019. <https://uavchallenge.org/about/sponsors-and-supporters/>.
- [4] 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.
- [5] E. S. M. Ebeid, M. Skriver, and J. Jin, "A survey on open-source flight control platforms of unmanned aerial vehicle," 08 2017.
- [6] "ArduPilot - strona domowa projektu," Dostęp z 2020. <https://ardupilot.org/>.
- [7] "Autoquad - historia projektu," Dostęp z 2020. <http://autoquad.org/home/autoquad-project-timeline/>.
- [8] "Librepilot - strona domowa projektu," Dostęp z 2020. <https://www.librepilot.org/site/index.html>.
- [9] "Px4 autopilot - strona domowa projektu," Dostęp z 2020. <https://px4.io/>.
- [10] "ArduPilot - dokumentacja funkcjonalności symulatora autopilota," Dostęp z 2020. <https://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html>.
- [11] "ArduPilot - dział dokumentacji poświęcony protokołowi mavlink," Dostęp z 2020. <https://ardupilot.org/dev/docs/mavlink-commands.html>.
- [12] "Dronekit - strona domowa projektu," Dostęp z 2020. <https://dronekit.io/>.
- [13] "Protocol buffers - strona domowa projektu," Dostęp z 2020. <https://developers.google.com/protocol-buffers>.
- [14] "Protokół websocket - specyfikacja," Dostęp z 2020. [tools.ietf.org/html/rfc6455](https://tools.ietf.org/html/rfc6455).

- 
- [15] “Biblioteka `asyncio` - dokumentacja,” Dostęp z 2020. <https://docs.python.org/3/library/asyncio.html>.
  - [16] 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.
  - [17] A. F. Joseph Redmon, “Yolov3: An incremental improvement,”
  - [18] “Dokumentacja systemu kontroli `git` - rozdział poświęcony submodułom,” Dostęp z 2020. <https://git-scm.com/book/en/v2/Git-Tools-Submodules>.
  - [19] “Dokumentacja platformy `Docker`,” Dostęp z 2020. <https://docs.docker.com/>.
  - [20] “Dokumentacja platformy `Docker` - dział poświęcony woluminom `docker volumes`,” Dostęp z 2020. <https://docs.docker.com/storage/volumes/>.
  - [21] “Dokumentacja platformy `Docker` - dział poświęcony narzędziu `docker compose`,” Dostęp z 2020. <https://docs.docker.com/compose/>.