Zaawansowane Środowiska Symulacyjne dla Autonomicznych Rojów Robotów: Strategia Rozwoju "Zero-Hardware"

1. Streszczenie Zarządcze i Imperatyw Strategiczny

Krajobraz operacyjny misji poszukiwawczo-ratowniczych (SAR) przechodzi fundamentalną transformację napędzaną integracją systemów autonomicznych. Projekt Roju Dronów (SDS – Swarm Drone System), obejmujący zwinne jednostki zwiadowcze ("Faza 1") oraz ciężkie platformy ewakuacyjne ("Faza 2"), stanowi wyzwanie inżynieryjne o wysokiej złożoności, które wymaga rygorystycznej metodologii rozwoju opartej na symulacji ("simulation-first").[1]

Ryzyko związane z wdrażaniem autonomicznych rojów – w szczególności tych z logistyką ciężkiego udźwigu zdolną do transportu ludzi – wyklucza iteracyjny cykl oparty wyłącznie na sprzęcie fizycznym. Koszty utraty płatowców, bezpieczeństwo personelu testowego oraz ograniczenia prawne (takie jak metodologia SORA w UE) wymagają, aby "cyfrowy bliźniak" (digital twin) systemu osiągnął dojrzałość, zanim zostaną uruchomione fizyczne prototypy.[1]

Niniejszy raport przedstawia kompleksową strategię rozpoczęcia prac nad systemem SAR w środowisku całkowicie wirtualnym. Ustanawia on ekosystem symulacyjny o wysokiej wierności, który jest **izomorficzny** względem docelowej architektury sprzętowej (NVIDIA Jetson Orin Nano + Pixhawk 6C).[1] Wykorzystując techniki Software-In-The-Loop (SITL), konteneryzację mikroserwisów oraz middleware komunikacyjny Eclipse Zenoh, ramy te pozwalają na walidację krytycznych zachowań roju – takich jak algorytmy stadne (Boids) i konsensus Raft – w warunkach realistycznych ograniczeń sieciowych.

Celem jest odseparowanie dojrzałości oprogramowania od dostępności sprzętu. Podejście to nie tylko przyspiesza harmonogram prac, umożliwiając równoległe działanie zespołów ds. wizji, sterowania i interfejsów, ale także zapewnia deterministyczne środowisko do testowania systemu w przypadkach brzegowych (awarie czujników, zerwanie łączności), które są trudne lub niebezpieczne do odtworzenia w terenie.

2. Architektura Izomorficzna: Strategia Cyfrowego Bliźniaka

Aby zapewnić, że oprogramowanie stworzone w symulacji będzie przenoszalne na fizyczny statek powietrzny z minimalnymi modyfikacjami, architektura symulacji musi ściśle przestrzegać zasady izomorfizmu. Każdy komponent oprogramowania, interfejs i kanał komunikacyjny obecny w fizycznym systemie musi mieć swój bezpośredni, funkcjonalny

odpowiednik w domenie wirtualnej.

2.1 Paradygmat Rozdzielonego Przetwarzania: Integracja NuttX i Linux

Fizyczny dron wykorzystuje kontroler lotu Holybro Pixhawk 6C z autopilotem PX4 (system czasu rzeczywistego NuttX), połączony przez UART z komputerem pokładowym NVIDIA Jetson Orin Nano (Linux Ubuntu).[1] Odzwierciedlenie tego podziału jest kluczowe, ponieważ opóźnienia i przepustowość łącza między kontrolerem lotu (FCU) a komputerem towarzyszącym (Companion Computer) często stają się wąskim gardłem systemu.

W proponowanym środowisku ten podział sprzętowy jest modelowany za pomocą separacji procesów i mostkowania sieciowego:

- **Wirtualny Kontroler Lotu (PX4 SITL):** Zamiast działać na mikrokontrolerze STM32, firmware PX4 jest kompilowane jako natywny plik binarny Linux (px4). Ten proces wykonuje dokładnie te same pętle estymacji (EKF2) i sterowania, co firmware wbudowany, ale działa na procesorze stacji roboczej.[5] Pozwala to na instancjonowanie dziesiątek wirtualnych kontrolerów na jednym serwerze, umożliwiając testowanie skalowalności roju bez zakupu sprzętu.[5]
- **Wirtualny Komputer Towarzyszący (Kontener Docker):** Stos oprogramowania przeznaczony dla Jetson Orin Nano – obejmujący węzły ROS 2, mostki Zenoh i silniki AI – jest zamknięty w kontenerach Docker. Każdemu dronowi w roju przypisany jest unikalny kontener, który funkcjonuje jako jego izolowany "komputer", z własną przestrzenią nazw sieciowych, systemem plików i adresem IP. Ta izolacja wymusza na programistach respektowanie rozproszonej natury systemu; kod działający w kontenerze "Drona 1" nie może arbitralnie czytać zmiennych z "Drona 2", co odzwierciedla fizyczną rzeczywistość.[7]

2.2 Silnik Fizyczny: Gazebo Harmonic

Wybór silnika symulacyjnego jest kluczowy dla Fazy 2 (Heavy Lift). Projekt wymaga symulacji platformy o masie startowej (MTOW) przekraczającej 100 kg, przenoszącej dynamiczny ładunek (człowieka na noszach).[1] Starsze symulatory często upraszczają aerodynamikę i dynamikę bryły sztywnej, prowadząc do "luki w rzeczywistości" (reality gap).

Dlatego ramy te określają **Gazebo Harmonic** (część nowego ekosystemu Gazebo Sim, dawniej Ignition) jako rdzeń fizyczny. Harmonic wykorzystuje domyślnie silnik fizyki DART, który oferuje doskonałą obsługę ciał przegubowych i łańcuchów kinematycznych – co jest niezbędne do modelowania wahadłowej dynamiki noszy podwieszonych pod dronem.[9]

2.3 Luka w Middleware: DDS vs. Zenoh

Standardowa symulacja ROS 2 domyślnie używa Data Distribution Service (DDS). Jednak projekt wyraźnie wymaga **Eclipse Zenoh**, aby zminimalizować narzut związany z ruchem "discovery" w sieciach bezprzewodowych.[1] Użycie standardowego DDS na localhost ukryłoby

wyzwania sieciowe, które Zenoh ma rozwiązać.

W rezultacie architektura symulacji wyraźnie wyłącza standardowe odkrywanie ROS 2 między symulowanymi dronami. Zamiast tego wymusza topologię pośredniczoną przez Zenoh. Każdy kontener drona uruchamia zenoh-bridge-ros2dds, który tuneluje lokalny ruch ROS 2 do innych dronów za pomocą protokołu Zenoh, wiernie modelując ograniczoną przepustowość sieci polowej.[12]

Tabela 1: Mapowanie Domeny Fizycznej na Wirtualną

Komponent Fizyczny	Odpowiednik w Symulacji	Uzasadnienie Wyboru
Kontroler Lotu (Pixhawk 6C)	**PX4 SITL** (Binary)	Uruchamia identyczny kod lotu; umożliwia skalowanie roju bez sprzętu. [5]
Komputer Pokładowy (Jetson Orin)	**Kontener Docker** (Ubuntu 22.04)	Zapewnia izolację procesów i zależności; emuluje oddzielną przestrzeń sieciową. [13]
Sensory (OAK-D, GPS, IMU)	**Wtyczki Gazebo Harmonic**	Generuje syntetyczne dane z modelami szumów i aktywną projekcją głębi. [14]
Komunikacja (Wi-Fi/LTE)	**Wirtualna Sieć + tc + Zenoh**	Emuluje topologię sieci, opóźnienia i utratę pakietów; waliduje wydajność Zenoh. [15]
Interfejs Wizualny (Tablet/GCS)	**Web Interface / QGroundControl**	Łączy się przez TCP/UDP z magistralą symulacji, walidując telemetrię. [6]

3. Budowa Środowiska Deweloperskiego

Wdrożenie tej architektury wymaga powtarzalnego środowiska ("Swarm-in-a-Box") opartego na Dockerze.

3.1 Strategia Kontenera Bazowego

Środowisko opiera się na wielowarstwowym pliku Dockerfile. Bazowym systemem operacyjnym jest **Ubuntu 22.04 LTS (Jammy Jellyfish)**, który jest platformą Tier 1 dla ROS 2

Humble i Gazebo Harmonic.[9]

Dockerfile musi orkiestrować instalację czterech stosów technologicznych:

1. **Robotyka:** ros-humble-desktop-full oraz ros-humble-rmw-cyclonedds-cpp.
2. **Symulacja:** gz-harmonic oraz ros-humble-ros-gz (mostek między fizyką a ROS 2).[17]
3. **Flight Stack:** Sklonowanie i budowa repozytorium PX4-Autopilot. Wymaga to specyficznego toolchaina arm-none-eabi.[7]
4. **Programowanie Systemowe:** Instalacja toolchaina **Rust** (rustup) oraz wtyczek colcon-cargo i colcon-ros-cargo. Umożliwia to kompilację logiki roju napisanej w Rust obok węzłów C++ i Python.[18]

Kluczowe jest skonfigurowanie kontenera z **NVIDIA Container Toolkit**, aby węzły AI (YOLOv8) miały dostęp do GPU hosta, co zapewnia działanie symulacji w czasie rzeczywistym (Real-Time Factor ~ 1.0).[20]

3.2 Konfiguracja Topologii Sieci

Aby symulować rój, nie możemy uruchamiać wszystkich węzłów na interfejsie pętli zwrotnej (loopback) hosta. Używając docker-compose, definiujemy niestandardową sieć mostkową (bridge network).

- **Dron 1:** 10.10.1.11
- **Dron 2:** 10.10.1.12
- **GCS (Stacja Naziemna):** 10.10.1.100

Wewnątrz każdego kontenera pliki XML dla DDS są skonfigurowane tak, aby ograniczyć komunikację tylko do localhost (127.0.0.1).[12] To wymusza, aby cała komunikacja między dronami przechodziła przez **Mostek Zenoh**, co skutecznie symuluje "air-gap" (fizyczną separację) między dronami.

3.3 Układ Przestrzeni Roboczej (Workspace)

Struktura katalogów ~/sar_swarm_ws/src musi obsługiwać hybrydowy kod (Rust/Python/C++):

```
sar_swarm_ws/
└── src/
├── px4_msgs # Definicje wiadomości PX4 (pliki.msg)
├── ros2_rust # Biblioteka klienta Rust (rclrs)
├── sar_swarm_control # [Nowe] Logika Roju w Rust
│ ├── Cargo.toml # Zależności Rust (raft-rs, zenoh, etc.)
│ └── src/main.rs # Implementacja Boids i Konsensusu
├── sar_perception # [Nowe] Węzeł Wizyjny w Pythonie
│ ├── setup.py
│ └── sar_perception/node.py
```

└── sar_simulation # [Nowe] Modele SDF, Światy i Skrypty startowe
Pakiet px4_msgs musi być zgodny z wersją firmware'u PX4 używaną w symulacji, aby zapewnić binarną kompatybilność uORB.[16]

4. Implementacja Middleware: Mostek Zenoh

W symulacji wdrażamy routowaną architekturę Zenoh, wykraczającą poza proste demo peer-to-peer.

4.1 Konfiguracja Mostka Zenoh

Każdy symulowany dron uruchamia instancję zenoh-bridge-ros2dds. Plik konfiguracyjny (JSON5/YAML) montowany w kontenerze realizuje dwie funkcje:

1. **Przestrzenie Nazw (Namespacing):** Prefiksowanie wszystkich wychodzących tematów ROS 2 unikalnym kluczem.
    - Dron 1: namespace: "/swarm/drone_1"
2. **Filtrowanie Ruchu (Allow/Deny Lists):** Jak zidentyfikowano w badaniach, ślepe mostkowanie wszystkich tematów nasyca łącze.[1] Konfiguracja musi jawnie zezwalać (allow) tylko na krytyczne tematy (np. pozycja, status baterii), blokując ciężkie dane jak surowe chmury punktów LIDAR.[21]

4.2 Symulacja Degradacji Sieci (Network Degradation)

Aby zwalidować algorytm konsensusu Raft (zaimplementowany w Rust), musimy wprowadzić sztuczne błędy sieciowe. Używamy narzędzia **Traffic Control (tc)** wewnątrz kontenerów Docker.

Komenda Implementacyjna:
tc qdisc add dev eth0 root netem delay 100ms 20ms distribution normal loss 5%
Symuluje to:

- **Opóźnienie:** 100ms (typowe dla łączy dalekiego zasięgu/LoRa).[24]
- **Jitter:** ±20ms (testuje synchronizację czasu).
- **Utratę Pakietów:** 5% (stress-test dla mechanizmu replikacji logów Raft i niezawodności Zenoh).[15]

Testowanie w tych warunkach pozwala zespołowi inżynierów Rust dostroić interwały *heartbeat* w bibliotece raft-rs (np. zwiększając je z 50ms do 500ms) przed wdrożeniem na rzeczywiste drony.[25]

5. Dynamika Lotu i Orkiestracja Wielu Pojazdów

5.1 Faza 1: Symulacja Zwiadowców (Scout Swarm)

Dla roju zwiadowczego używamy modelu ramy **Holybro X500**. Uruchomienie instancji w roju odbywa się poprzez zmienne środowiskowe PX4:

Bash

```
PX4_SYS_AUTOSTART=4001 PX4_GZ_MODEL_POSE="0,0,0"./build/px4_sitl_default/bin/px4 -i 0
```

Skrypt powłoki iteruje przez ID instancji (-i 1, -i 2), uruchamiając kolejne procesy px4 i modele Gazebo w przesuniętych współrzędnych.[26]

5.2 Faza 2: Dynamika Heavy Lift

Platforma Heavy Lift (oktokopter >100kg) wymaga niestandardowego modelu SDF w Gazebo Harmonic:

1. **Macierz Bezwładności:** Precyzyjne obliczenie momentów I_{xx}, I_{yy}, I_{zz} dla masy 100kg, aby uniknąć niestabilnych oscylacji.
2. **Dynamika Ładunku:** Użycie wtyczki **Detachable Joint**. Nosze są modelowane jako oddzielne ciało sztywne (80kg) połączone z dronem przegubem kulowym. Symuluje to efekt wahadła, co pozwala zespołowi sterowania na dostrojenie algorytmów tłumienia (Input Shaping) w kontrolerze Rust.[18]

6. Symulacja Sensorów i Percepcji

6.1 Symulacja Aktywnej Głębi (OAK-D)

Odtwarzamy funkcjonalność kamery Luxonis OAK-D Pro w Gazebo Harmonic poprzez kombinację sensorów:

- **Kamery Stereo:** Para kamer RGB oddalona o bazę 7.5cm.[29]
- **Aktywny Projektor:** Użycie wtyczki **Projector Plugin** w Gazebo, która rzuca teksturę (wzór plamek) na świat symulacji. Pozwala to algorytmom stereo (uruchomionym w ROS) na generowanie map głębi nawet na gładkich powierzchniach, wiernie replikując zalety oświetlacza IR w prawdziwym sprzęcie.[30]

6.2 Luka Programowo-Sprzętowa: Latencja

W symulacji wnioskowanie AI (detekcja ludzi) odbywa się na GPU hosta (np. 5ms), podczas gdy na OAK-D trwa to 30-50ms. Aby zachować izomorfizm, węzeł percepcji (Python) musi celowo wprowadzać opóźnienie (np. time.sleep(0.04)), aby zweryfikować, czy logika sterowania jest odporna na opóźnienia, które w rzeczywistości mogą powodować oscylacje

(PIO - Pilot Induced Oscillation).[17]

7. "Mózg" Roju: Implementacja w Rust

7.1 Rozproszony Konsensus (Raft-rs)

Rój musi uzgodnić stan misji. Używamy biblioteki **raft-rs**. Każdy dron uruchamia węzeł Raft.

- **Testowanie:** Symulujemy "Awarię Lidera" poprzez zatrzymanie kontenera Docker (docker stop drone_1). Weryfikujemy w logach pozostałych dronów, czy nowa elekcja lidera nastąpiła w zadanym oknie czasowym.[25]

7.2 Sterowanie Stadne (Boids)

Algorytm Boids (Separacja, Wyrównanie, Spójność) jest zaimplementowany jako czysta funkcja w Rust.

- Wejście: Pozycje sąsiadów (przez Zenoh).
- Wyjście: Wektor prędkości publikowany jako TrajectorySetpoint do PX4.33
  Wizualizacja wektorów sił (np. siły separacji) w RViz pozwala na intuicyjne debugowanie zachowań roju i zapobieganie kolizjom.

8. Harmonogram Wdrożenia (Roadmapa)

Faza 1: Fundament (Tygodnie 1-3)

- Instalacja Ubuntu 22.04, Docker, NVIDIA Container Toolkit.
- Stworzenie Dockerfile (ROS 2 Humble + Rust + PX4).
- Pierwszy lot wirtualny (SITL) sterowany przez QGroundControl.

Faza 2: Rój (Tygodnie 4-6)

- Skrypt uruchamiający wiele kontenerów z unikalnymi Namespaces.
- Implementacja węzła sterującego w Rust (logika Boids).
- Stress-testy sieciowe (użycie tc do symulacji utraty pakietów).

Faza 3: Misja (Tygodnie 7-10)

- Modyfikacja modelu SDF (dodanie symulacji OAK-D i projektora).
- Wdrożenie węzła wizyjnego (Python/YOLO) z symulowaną latencją.
- Pełna misja: Zwiad $\rightarrow$ Detekcja $\rightarrow$ Konsensus $\rightarrow$ Konwergencja na celu.

9. Wnioski

Droga do funkcjonalnego roju SAR nie zaczyna się na pasie startowym, lecz w laboratorium symulacyjnym. Budując środowisko "Digital Twin" opisane w tym raporcie, zespół może

zminimalizować ryzyko związane z ciężkimi systemami autonomicznymi. Ramy te, oparte na filarach **izomorfizmu** (PX4 SITL), **łączności** (Zenoh) i **bezpieczeństwa** (Rust), przekształcają złożoność robotyki roju w zarządzalne wyzwanie inżynierii oprogramowania. Zapewnia to, że w momencie uruchomienia fizycznych wirników, będą one sterowane przez logikę, która przetrwała już chaos w świecie wirtualnym.

Works cited

1. Projekt Dronów SAR: Ewakuacja Człowieka
2. ROS 2 and Zenoh – From Zero to Deployment - YouTube, accessed on January 1, 2026, https://www.youtube.com/watch?v=-WZAA35jLxc
3. How to setup PX4 SITL with ROS2 and XRCE-DDS Gazebo simulation on Ubuntu 22, accessed on January 1, 2026, https://kuat-telegenov.notion.site/How-to-setup-PX4-SITL-with-ROS2-and-XRCE-DDS-Gazebo-simulation-on-Ubuntu-22-e963004b701a4fb2a133245d96c4a247
4. Installation with Gazebo Classic - PX4 Discussion Forum, accessed on January 1, 2026, https://discuss.px4.io/t/installation-with-gazebo-classic/40374
5. Simulation | PX4 Guide (main), accessed on January 1, 2026, https://docs.px4.io/main/en/simulation/
6. Multi-Vehicle Simulation with JMAVSim | PX4 Guide (main), accessed on January 1, 2026, https://docs.px4.io/main/en/sim_jmavsim/multi_vehicle
7. PX4 Docker Containers | PX4 Guide (main), accessed on January 1, 2026, https://docs.px4.io/main/en/test_and_ci/docker
8. Building a Professional PX4 Development Environment with Docker, ROS2, and VS Code, accessed on January 1, 2026, https://dev.to/james-odukoya/building-a-professional-px4-development-environment-with-docker-ros2-and-vs-code-bgo
9. Gazebo Simulation | PX4 Guide (main), accessed on January 1, 2026, https://docs.px4.io/main/en/sim_gazebo_gz/
10. Gazebo Harmonic Released! - Open Robotics, accessed on January 1, 2026, https://www.openrobotics.org/blog/2023/9/26/gazebo-harmonic-released
11. artastier/PX4_Swarm_Controller: The aim of this ROS2 package is to facilitate the implementation of a drone swarm controller through simulation on Gazebo. - GitHub, accessed on January 1, 2026, https://github.com/artastier/PX4_Swarm_Controller
12. Use zenoh-bridge-ros2dds with ROS2 Humble | by William Chen - Medium, accessed on January 1, 2026, https://medium.com/@piliwilliam0306/use-zenoh-bridge-ros2dds-with-ros2-humble-459ab70ce9c7
13. Setup ROS 2 with VSCode and Docker [community-contributed] - ROS documentation, accessed on January 1, 2026, https://docs.ros.org/en/iron/How-To-Guides/Setup-ROS-2-with-VSCode-and-Docker-Container.html
14. Sensors — Gazebo jetty documentation, accessed on January 1, 2026, https://gazebosim.org/docs/latest/sensors/

15. Simulate high latency network using Docker containers and “tc” commands - Medium, accessed on January 1, 2026, https://medium.com/@kazushi/simulate-high-latency-network-using-docker-containerand-tc-commands-a3e503ea4307
16. ROS 2 User Guide | PX4 Guide (main), accessed on January 1, 2026, https://docs.px4.io/main/en/ros2/user_guide
17. DepthAI ROS - Luxonis Docs, accessed on January 1, 2026, https://docs.luxonis.com/software/ros/depthai-ros/
18. ros2-rust/ros2_rust: Rust bindings for ROS 2 - GitHub, accessed on January 1, 2026, https://github.com/ros2-rust/ros2_rust
19. The first steps when using Rust with ROS 2 - Foxglove, accessed on January 1, 2026, https://foxglove.dev/blog/first-steps-using-rust-with-ros2
20. mzahana/px4_ros2_humble: A Docker development environment for PX4 + ROS 2 Humble, accessed on January 1, 2026, https://github.com/mzahana/px4_ros2_humble
21. Zenoh with turtlebot4 - ros2 - Robotics Stack Exchange, accessed on January 1, 2026, https://robotics.stackexchange.com/questions/105219/zenoh-with-turtlebot4
22. PX4/px4_msgs: ROS/ROS2 messages that match the uORB messages counterparts on the PX4 Firmware - GitHub, accessed on January 1, 2026, https://github.com/PX4/px4_msgs
23. [Bug] Denying/allowing topics on the subscriber side does not work as expected #241 - GitHub, accessed on January 1, 2026, https://github.com/eclipse-zenoh/zenoh-plugin-ros2dds/issues/241
24. Manage Quality of Service Policies in ROS 2 Application with TurtleBot - MathWorks, accessed on January 1, 2026, https://www.mathworks.com/help/ros/ug/manage-quality-of-service-policies-using-ros2-in-turtlebot.html
25. Implementing a distributed ticket booking service in Rust using The Raft Consensus Algorithm - Disant Upadhyay, accessed on January 1, 2026, https://disant.medium.com/implementing-a-distributed-ticket-booking-service-in-rust-using-the-raft-consensus-algorithm-b04c1305f3ce
26. Multi-Vehicle Simulation with Gazebo - PX4/PX4-user_guide - GitHub, accessed on January 1, 2026, https://github.com/PX4/PX4-user_guide/blob/main/tr/sim_gazebo_gz/multi_vehicle_simulation.md
27. Multi Vehicle Gazebo SITL port configuration - PX4 Discussion Forum, accessed on January 1, 2026, https://discuss.px4.io/t/multi-vehicle-gazebo-sitl-port-configuration/32753
28. ROS2_Control Gazebo Tutorial Robot Simulation (Custom Robot Arm TeslaBot!) - YouTube, accessed on January 1, 2026, https://www.youtube.com/watch?v=PM_1Nb9u-N0
29. OAK-D Pro - Luxonis Docs, accessed on January 1, 2026, https://docs.luxonis.com/hardware/products/OAK-D%20Pro
30. Simulating a Structured Light Sensor in Gazebo, accessed on January 1, 2026,

https://answers.gazebosim.org/question/16441/
31. Tutorial : Gazebo plugins in ROS, accessed on January 1, 2026, https://classic.gazebosim.org/tutorials?tut=ros_gzplugins
32. tikv/raft-rs: Raft distributed consensus algorithm implemented in Rust. - GitHub, accessed on January 1, 2026, https://github.com/tikv/raft-rs
33. superit23/SwarmRobotics: Python implementation of Boid flocking algorithm for ROS, accessed on January 1, 2026, https://github.com/superit23/SwarmRobotics
34. boids_rs_bevy - crates.io: Rust Package Registry, accessed on January 1, 2026, https://crates.io/crates/boids_rs_bevy