Advanced Simulation Frameworks for Autonomous Swarm Robotics: A Comprehensive Development Strategy for Non-Hardware Prototyping

1. Executive Summary and Strategic Imperative

The operational landscape of Search and Rescue (SAR) is undergoing a fundamental transformation, driven by the integration of autonomous systems capable of operating in unstructured, hazardous environments. The project under analysis—a Swarm Drone System (SDS) encompassing agile reconnaissance units ("Phase 1") and heavy-lift extraction platforms ("Phase 2")—represents a high-complexity engineering challenge that necessitates a rigorous, simulation-first development methodology.[1] The inherent risks associated with deploying autonomous swarms, particularly those involving heavy-lift logistics capable of transporting human-scale payloads, preclude a hardware-centric iterative cycle. The financial cost of airframe loss, the safety implications for test personnel, and the regulatory constraints imposed by aviation authorities (such as the Specific Operations Risk Assessment or SORA in the EU) demand that the system's "digital twin" be mature before physical prototypes are engaged.[1]

This report articulates a comprehensive strategy for initiating the development of this SAR system in a purely virtual environment. It establishes a high-fidelity simulation ecosystem that is isomorphic to the target hardware architecture, specifically the NVIDIA Jetson Orin Nano companion computers and Pixhawk 6C flight controllers.[1] By leveraging Software-In-The-Loop (SITL) techniques, containerized microservices, and the Eclipse Zenoh communication middleware, this framework allows for the validation of critical swarm behaviors—such as Boids-based flocking and Raft-based consensus—under realistic network constraints.

The objective is to decouple software maturity from hardware availability. This approach not only accelerates the development timeline by allowing parallel workstreams for vision, control, and interface teams but also provides a deterministic environment for stress-testing the system against edge cases—network partitions, sensor failures, and environmental disturbances—that are difficult or dangerous to reproduce in field trials. This document serves as the foundational technical roadmap for constructing this "Zero-Hardware" development environment, ensuring that the transition to physical flight is a validation of proven logic rather than an experiment in stability.

2. Architectural Isomorphism: The Digital Twin

Strategy

To ensure that software developed in simulation is portable to the physical aircraft with minimal modification, the simulation architecture must rigidly adhere to the principle of isomorphism. Every software component, interface, and communication channel present in the physical system must have a direct, functional equivalent in the virtual domain. The target hardware architecture, defined by a split-compute model utilizing a flight controller for real-time stabilization and a powerful companion computer for high-level autonomy, dictates the structure of the simulation.

2.1 The Split-Compute Paradigm: NuttX and Linux Integration

The physical drone employs a Holybro Pixhawk 6C running the PX4 Autopilot on a real-time operating system (NuttX), connected via UART/serial to an NVIDIA Jetson Orin Nano running Ubuntu Linux.[1] Replicating this separation is critical because the latency and bandwidth constraints of the link between the flight controller (FCU) and the companion computer often become system bottlenecks.

In the proposed simulation environment, this hardware split is modeled using process separation and network bridging:

- **Virtual Flight Controller (PX4 SITL):** Instead of running on the STM32H7 microcontroller of the Pixhawk, the PX4 firmware is compiled as a native Linux binary (px4). This binary runs the exact same estimation and control loops (EKF2, rate controllers) as the embedded firmware but executes on the host workstation's CPU.[5] This is distinct from Hardware-In-The-Loop (HITL), which would require physical Pixhawks on the desk; SITL allows for the instantiation of dozens of virtual flight controllers on a single server, enabling swarm scalability testing without hardware procurement.[5]
- **Virtual Companion Computer (Docker Container):** The software stack intended for the Jetson Orin Nano—comprising ROS 2 nodes, Zenoh bridges, and AI inference engines—is encapsulated within Docker containers. Each drone in the swarm is assigned a unique container that functions as its isolated "computer," complete with its own network namespace, file system, and IP address. This isolation forces developers to respect the distributed nature of the system; code running in "Drone 1's" container cannot arbitrarily access memory variables in "Drone 2's" container, mirroring the physical reality.[7]

2.2 The Physics Engine: Gazebo Harmonic

The choice of simulation engine is pivotal for the Heavy Lift phase (Phase 2). The project requires simulating a multi-rotor platform with a Maximum Take-Off Weight (MTOW) exceeding 100 kg, carrying a suspended dynamic load (the evacuee).[1] Legacy simulators often simplify aerodynamic drag and rigid body dynamics, leading to a "reality gap" where controllers tuned in simulation oscillate violently in the real world.

Therefore, this framework specifies **Gazebo Harmonic** (part of the new Gazebo Sim ecosystem, formerly Ignition) as the physics core. Harmonic utilizes the DART (Dynamic Animation and Robotics Toolkit) physics engine by default, which offers superior handling of articulated bodies and kinematic chains—essential for modeling the pendulum dynamics of a stretcher suspended beneath a heavy-lift drone.[9] Furthermore, Harmonic's architecture separates the physics server (gz-server) from the rendering client (gz-client), allowing for headless execution of massive swarms on cloud servers for continuous integration (CI) testing.[11]

2.3 The Middleware Gap: DDS vs. Zenoh

A standard ROS 2 simulation defaults to the Data Distribution Service (DDS) for communication. However, the project explicitly mandates **Eclipse Zenoh** to mitigate the discovery traffic overhead inherent to DDS in wireless mesh networks.[1] If the simulation uses standard DDS on localhost, it will fail to expose the networking challenges that Zenoh is designed to solve.

Consequently, the simulation architecture explicitly disables standard ROS 2 discovery between simulated drones. Instead, it enforces a Zenoh-mediated topology. Each drone container runs a zenoh-bridge-ros2dds that encapsulates local ROS 2 traffic and tunnels it to other drones via the Zenoh protocol (tcp/udp), accurately modeling the bandwidth-constrained, routed network of the field deployable system.[13]

Table 1 summarizes the mapping between the physical and virtual domains, establishing the blueprint for the development environment.

Physical Component	**Simulated Counterpart**	**Justification for Selection**
Flight Controller (Pixhawk 6C)	**PX4 SITL** (Binary)	Runs identical flight code; enables scalable instantiation of multiple agents without hardware. [5]
Companion Computer (Jetson Orin)	**Docker Container** (Ubuntu 22.04)	Provides process isolation and dependency management; emulates the network namespace of a distinct device. [14]
Sensors (OAK-D, GPS,	**Gazebo Harmonic Plugins**	Generates synthetic sensor

IMU)		data with configurable noise models and active depth projection. [15]
Communication (Wi-Fi/LTE)	**Virtual Network + tc + Zenoh**	Emulates network topology, latency, and packet loss; validates Zenoh's efficiency over DDS. [16]
Visual Interface (Tablet/GCS)	**Web Interface / QGroundControl**	Connects via TCP/UDP to the simulation bus, validating telemetry streams and command latency. [6]

3. Construction of the Development Environment

The implementation of this architecture requires a robust, reproducible development environment. We utilize Docker to create a portable "Swarm-in-a-Box" solution that allows developers to spin up the entire infrastructure on a standard laptop or a cloud instance.

3.1 The Base Container Strategy

The development environment relies on a tiered Dockerfile strategy to manage the complex dependency trees of ROS 2 Humble, Rust, and PX4. The base operating system is **Ubuntu 22.04 LTS (Jammy Jellyfish)**, which is the Tier 1 platform for ROS 2 Humble and Gazebo Harmonic.[9]

The **Dockerfile** must orchestrate the installation of four distinct technology stacks:

1. **Robotics Middleware:** Installation of ros-humble-desktop-full and ros-humble-rmw-cyclonedds-cpp to provide the core messaging backbone.
2. **Simulation Layer:** Installation of gz-harmonic and the ros-gz bridge packages (ros-humble-ros-gz) to facilitate data exchange between the physics engine and ROS 2.[18]
3. **Flight Stack:** Cloning and building the PX4-Autopilot repository. This requires the installation of the specific arm-none-eabi toolchain and Python dependencies for the uORB code generation.[7]
4. **Systems Programming:** Installation of the Rust toolchain (rustup), alongside the colcon-cargo and colcon-ros-cargo build plugins. This enables the standard ROS build tool (colcon) to compile the Rust-based swarm logic alongside C++ and Python nodes seamlessly.[19]

Crucially, to support the AI requirements (Phase 1 Perception), the container must be

configured with the **NVIDIA Container Toolkit**. This allows the containerized PyTorch/YOLOv8 nodes to access the host's GPU for inference, ensuring that the simulation runs at real-time factors (RTF) close to 1.0, which is essential for accurate flight dynamics.[21]

3.2 Network Topology Configuration

To simulate a swarm, we cannot simply run all nodes on the host's loopback interface. We must create a virtual network topology that enforces the routing logic of Zenoh.

Using docker-compose, we define a custom bridge network. Each drone service is assigned a static IP address within a specific subnet (e.g., 10.10.1.0/24).

- **Drone 1:** 10.10.1.11
- **Drone 2:** 10.10.1.12
- **GCS:** 10.10.1.100

Within each container, the ROS_DOMAIN_ID is used to logically isolate the ROS 2 traffic. However, since ROS 2 discovery (DDS) operates via multicast, simply changing the Domain ID is not enough to simulate *network* isolation if they share the same subnet. To rigorously test Zenoh, we configure the DDS XML profiles within each container to restrict communication to localhost (127.0.0.1) only.[13] This forces all inter-drone communication to pass through the **Zenoh Bridge**, effectively simulating the air-gap between physical drones.

3.3 Workspace Layout for Polyglot Development

The project requires a hybrid software stack: Rust for control and Python for vision. The ROS 2 workspace (~/sar_swarm_ws/src) must be structured to accommodate both build systems.

The recommended structure is as follows:

```
sar_swarm_ws/
└── src/
├── px4_msgs # [Cloned] PX4 Message definitions (IDL/msg files)
├── ros2_rust # [Cloned] Rust Client Library (rclrs)
├── sar_swarm_control # [New] Rust-based Swarm Logic
│ ├── Cargo.toml # Rust dependencies (raft-rs, zenoh, etc.)
│ └── src/main.rs # Boids & Consensus implementation
├── sar_perception # [New] Python-based Vision Node
│ ├── setup.py
│ └── sar_perception/node.py
└── sar_simulation # [New] SDF Models, Worlds, and Launch Scripts
```

This layout ensures that colcon build can resolve the dependencies between the languages. The px4_msgs package is particularly critical; it must match the git hash of the PX4 firmware used in the simulation to ensure that the uORB topic serialization is binary-compatible.17

4. Middleware Implementation: The Zenoh Bridge

The implementation of Zenoh is the linchpin of the swarm's scalability. In the simulation, we move beyond the standard "peer-to-peer" demo and implement a routed architecture that reflects the project's operational reality.

4.1 Configuring the Zenoh Bridge

Each simulated drone runs an instance of zenoh-bridge-ros2dds. This binary acts as the gateway between the drone's internal ROS 2 network and the external Zenoh swarm network.

The bridge configuration is defined in a JSON5 or YAML file, which is mounted into the Docker container. This configuration performs two critical functions: **Namespacing** and **Traffic Filtering**.

Namespacing: To distinguish "Drone 1" from "Drone 2" in the global dataspace, the bridge is configured to prefix all outgoing ROS 2 topics with a unique namespace key.

- **Drone 1 Config:** namespace: "/swarm/drone_1"
- **Drone 2 Config:** namespace: "/swarm/drone_2"

Traffic Filtering (Allow/Deny Lists): As identified in the project research, blindly bridging all topics leads to saturation.[1] The configuration file must explicitly allow only mission-critical topics.

Code snippet

```
// zenoh_bridge_config.json5
{
  ros2dds: {
    namespace: "/swarm/drone_1",
    allow: {
      publishers:,
      subscribers:
    }
  }
}
```

This configuration ensures that high-bandwidth internal topics, such as raw LIDAR point clouds (/scan) or intermediate image buffers, are **never** transmitted over the simulated wireless link, verifying the bandwidth-saving architecture.[22]

4.2 Simulating Network Degradation

A "perfect" simulation on a local machine can mask critical failures in distributed consensus algorithms. The Raft algorithm, used for swarm state consistency, relies on strict timeouts. If network latency exceeds the election_timeout, the cluster will become unstable, triggering endless leader elections.

To validate the Rust implementation of Raft, we must introduce artificial network degradation. We utilize the Linux **Traffic Control (tc)** utility within the Docker containers. By manipulating the kernel's packet scheduler on the virtual Ethernet interface (eth0), we can inject precise fault models.

Implementation Command:
tc qdisc add dev eth0 root netem delay 100ms 20ms distribution normal loss 5%
This command introduces:

- **Latency:** 100ms base delay, mimicking a long-range LoRa or congested LTE link.[25]
- **Jitter:** ±20ms variation, testing the robustness of the time synchronization.
- **Packet Loss:** 5% random drop rate, which is the ultimate stress test for the Zenoh reliable reliability setting and the Raft log replication mechanism.[16]

Testing the swarm logic under these conditions *before* deployment is the single most valuable outcome of this simulation framework. It allows the Rust engineering team to tune the raft-rs heartbeat intervals (e.g., increasing them from 50ms to 500ms) to accommodate the reality of aerial networks.[26]

5. Flight Dynamics and Multi-Vehicle Orchestration

The simulation must accurately replicate the flight characteristics of the specific airframes chosen for the project.

5.1 Phase 1: Agile Scout Simulation

The **Holybro X500** is the designated platform for the scout swarm.[1] PX4 provides a verified parameter set for this frame. To launch a specific instance in the swarm, we use the PX4 startup environment variables:

Bash

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

- PX4_SYS_AUTOSTART=4001: Loads the X500 airframe mixer and PID gains.

- -i 0: Sets the instance ID to 0. This configures the MAVLink ports to 14540 (UDP) and the DDS/Zenoh bridge to look for specific namespaces.[9]

For the swarm, we create a shell script that iterates through instance IDs, spawning multiple px4 processes and their corresponding Gazebo models at offset coordinates (e.g., PX4_GZ_MODEL_POSE="0,2,0" for the second drone).[28]

5.2 Phase 2: Heavy Lift Dynamics

The heavy lift platform (Phase 2) presents a unique physics challenge. It is an **Octocopter (X8 configuration)** with a mass >100kg. Standard quadcopter models in simulations are often too "stiff" and responsive.

To simulate this accurately in Gazebo Harmonic, we must define a custom SDF (Simulation Description Format) model. This involves:

1. **Inertia Matrix:** accurately calculating the moments of inertia (I_{xx}, I_{yy}, I_{zz}) for a 100kg distributed mass. Errors here will cause the simulated drone to tumble or oscillate.
2. **Motor Model:** Configuring the motor plugins to reflect the slower spin-up time (time constant) of large heavy-lift motors (e.g., T-Motor U13).
3. **Payload Dynamics:** Using a **Detachable Joint** plugin in Gazebo to model the payload. The stretcher is simulated as a separate rigid body with mass (80kg) connected to the drone via a "Revolute" or "Ball" joint. This simulates the pendulum effect, which introduces complex destabilizing forces on the drone. This allows the control team to implement and tune "Input Shaping" or active damping algorithms in the Rust controller to mitigate load swing.[19]

6. Sensor Simulation and Perception Pipeline

The "eyes" of the swarm rely on the **Luxonis OAK-D Pro**, a sophisticated sensor fusing active stereo depth, RGB vision, and on-chip AI. Simulating this sensor is complex because the physical device processes data internally (VPU), whereas the simulation must process it on the host GPU.

6.1 Simulating Active Stereo Depth

Gazebo Harmonic does not have a "Luxonis" plugin. We recreate the sensor's functionality by combining standard Gazebo sensors:

- **RGB Camera:** A standard camera sensor publishing 1080p images.
- **Stereo Depth:** A pair of camera sensors separated by the OAK-D's baseline (7.5cm).[30]
- **Active Projector:** The OAK-D Pro uses an IR dot projector to assist depth perception on textureless surfaces (like snow or concrete walls). We simulate this using the Gazebo **Projector Plugin**, which projects a texture (a static speckle pattern) into the simulation

world. The simulated stereo cameras "see" this pattern, allowing the stereo matching algorithms (running in ROS) to generate dense depth maps even on flat white surfaces, faithfully replicating the hardware's advantage.[31]

6.2 The Software-Hardware Gap: Perception Latency

On the real drone, the OAK-D outputs object detections directly. In simulation, we must run the inference on the host. We use a Python node (sar_perception) subscribing to the Gazebo camera topic.
To maintain isomorphism, we must simulate the inference latency. The OAK-D might take 30-50ms to process a frame. The host GPU (e.g., RTX 4090) might do it in 5ms. If we feed 5ms latency data to the flight controller, the simulation will be optimistically stable.

- **Validation Strategy:** The Python node must explicitly introduce a delay (e.g., time.sleep(0.04)) before publishing detections to the /perception/human_found topic. This ensures that the Rust control logic is robust to the delay-induced instability that often causes "pilot-induced oscillation" (PIO) in visual servoing tasks.[18]

7. The Swarm "Brain": Rust Implementation

The core intelligence of the system—the Boids flocking and Raft consensus—is implemented in Rust. This selection is driven by the need for type safety and thread safety in complex distributed systems.

7.1 Distributed Consensus with Raft-rs

The swarm must agree on the mission state (e.g., "Search" vs. "Rescue"). We use the raft-rs crate, a production-grade implementation of the Raft consensus algorithm used in databases like TiKV.33
In the simulation, every drone runs a Raft node.

- **Storage:** We use sled (an embedded database in Rust) or an in-memory store for the Raft log, simulating the persistence layer.[34]
- **Transport:** Raft messages (AppendEntries, RequestVote) are serialized (using Protocol Buffers or Serde) and published to a specific ROS 2 topic (e.g., /swarm/raft_rpc) which is bridged via Zenoh to all other drones.
- **Testing Scenario:** We simulate a "Leader Failure" by simply stopping the Docker container of the current leader drone (docker stop drone_1). We then observe the logs of the remaining drones to verify that a new election is triggered and a new leader is established within the election_timeout window, validating the system's redundancy.[26]

7.2 Flocking Control (Boids)

The Boids algorithm calculates velocity vectors to maintain formation. This is implemented as a pure function in Rust for maximum performance.

- **Inputs:** A list of neighbor positions (received via Zenoh/ROS 2).

- **Vector Math:**
  - *Separation:* $\vec{v}_{sep} = \sum_{j \in neighbors} \frac{\vec{p}_i - \vec{p}_j}{||\vec{p}_i - \vec{p}_j||^2}$ (Repel from close neighbors).
  - *Alignment:* $\vec{v}_{align} = \frac{1}{N} \sum \vec{v}_j - \vec{v}_i$ (Match velocity).
  - *Cohesion:* $\vec{v}_{coh} = \frac{1}{N} \sum \vec{p}_j - \vec{p}_i$ (Move to centroid).
- **Output:** The weighted sum of these vectors is converted into a TrajectorySetpoint message and published to /fmu/in/trajectory_setpoint, which PX4 uses to drive the motors.[35]

By visualizing these vectors in **RViz** (the ROS visualization tool), developers can intuitively debug the swarm behavior, ensuring that the "Separation" force is sufficient to prevent mid-air collisions during high-speed maneuvers.

8. Implementation Roadmap

This report outlines a sequential execution plan to build this environment.

Phase 1: The Foundation (Weeks 1-3)

Objective: Establish a single simulated drone with full connectivity.

1. **Infrastructure:** Install Ubuntu 22.04, Docker, and NVIDIA Container Toolkit.
2. **Container Build:** Create the Dockerfile integrating ROS 2 Humble, Rust, and PX4 source.
3. **Single Flight:** Validate make px4_sitl gz_x500 launches a drone that can be controlled via QGroundControl.
4. **Zenoh Bridge:** Configure zenoh-bridge-ros2dds and verify that topics are visible on the host machine using the zenoh-cli tool.

Phase 2: The Swarm (Weeks 4-6)

Objective: Multi-vehicle orchestration and consensus.

1. **Swarm Launch Script:** Develop a script to launch 3+ Docker containers with unique IDs and namespaces.[27]
2. **Rust Control Node:** Implement the rclrs node with basic Boids logic.
3. **Network Stress Test:** Apply tc latency rules and verify that Raft consensus remains stable. If unstable, tune the heartbeat parameters in raft-rs.

Phase 3: The Mission (Weeks 7-10)

Objective: Full SAR loop with Perception.

1. **Sensor Upgrade:** Modify the X500 SDF to include the depth camera and projector plugin.
2. **Vision Node:** Deploy the Python YOLOv8 node with simulated latency.
3. **Integration:** Execute a full mission: Swarm Search $\rightarrow$ Detection (Python)

$\rightarrow$ Consensus (Rust) $\rightarrow$ Converge on Target.

9. Conclusion

The path to a functional Autonomous SAR Swarm does not begin on the runway; it begins in the simulation lab. By constructing the "Digital Twin" environment detailed in this report, the development team can mitigate the profound risks associated with heavy-lift autonomous systems. This framework, built on the pillars of **isomorphism** (PX4 SITL), **connectivity** (Zenoh), and **safety** (Rust), transforms the daunting complexity of swarm robotics into a manageable, testable software engineering challenge. It ensures that when the physical rotors finally spin, they are driven by logic that has already survived the chaos of the virtual world.

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. Getting Started with Gazebo? — Gazebo jetty documentation, accessed on January 1, 2026, https://gazebosim.org/docs/latest/getstarted/
12. 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
13. 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
14. 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
15. Sensors — Gazebo jetty documentation, accessed on January 1, 2026, https://gazebosim.org/docs/latest/sensors/
16. 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
17. ROS 2 User Guide | PX4 Guide (main), accessed on January 1, 2026, https://docs.px4.io/main/en/ros2/user_guide
18. DepthAI ROS - Luxonis Docs, accessed on January 1, 2026, https://docs.luxonis.com/software/ros/depthai-ros/
19. ros2-rust/ros2_rust: Rust bindings for ROS 2 - GitHub, accessed on January 1, 2026, https://github.com/ros2-rust/ros2_rust
20. 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
21. 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
22. Zenoh with turtlebot4 - ros2 - Robotics Stack Exchange, accessed on January 1, 2026, https://robotics.stackexchange.com/questions/105219/zenoh-with-turtlebot4
23. 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
24. [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
25. 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
26. 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
27. 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

28. 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
29. 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
30. OAK-D Pro - Luxonis Docs, accessed on January 1, 2026, https://docs.luxonis.com/hardware/products/OAK-D%20Pro
31. Simulating a Structured Light Sensor in Gazebo, accessed on January 1, 2026, https://answers.gazebosim.org/question/16441/
32. Tutorial : Gazebo plugins in ROS, accessed on January 1, 2026, https://classic.gazebosim.org/tutorials?tut=ros_gzplugins
33. tikv/raft-rs: Raft distributed consensus algorithm implemented in Rust. - GitHub, accessed on January 1, 2026, https://github.com/tikv/raft-rs
34. TheDhejavu/raft-consensus: Rust Implementation of the Raft distributed consensus protocol (https://raft.github.io), accessed on January 1, 2026, https://github.com/TheDhejavu/raft-consensus
35. superit23/SwarmRobotics: Python implementation of Boid flocking algorithm for ROS, accessed on January 1, 2026, https://github.com/superit23/SwarmRobotics
36. boids_rs_bevy - crates.io: Rust Package Registry, accessed on January 1, 2026, https://crates.io/crates/boids_rs_bevy