WYKŁAD 12

Programowanie autonomicznych robotów

Programowanie aplikacji mobilnych
dr inż. Mateusz Pomianek

ROS 2 · SLAM · Python/C++ · Gazebo · Nav2 · MoveIt

- Architektura systemów i ROS 2
- Percepcja: LiDAR, kamera, SLAM
- Planowanie ścieżki i nawigacja
- Sterowanie: PID, BT, RL
- Bezpieczeństwo i standardy

Plan wykładu

Sl. 1–3
Autonomia, architektury i ekosystem ROS 2

Sl. 4–5
Kinematyka - DH, forward/inverse kinematics

Sl. 6–7
ROS 2 - węzły, topiki, serwisy, akcje (Python/C++)

Sl. 8–9
Sensory robotów - LiDAR, stereo, IMU, enkodery

Sl. 10–11
Percepcja 3D - chmury punktów, PCL, YOLO dla robotów

Sl. 12–13
SLAM - Simultaneous Localization and Mapping

Sl. 14–15
Planowanie ścieżki - A*, RRT*, Dijkstra, Nav2

Sl. 16–17
Sterowanie - PID, maszyny stanów, Behavior Trees

Sl. 18–19
MoveIt 2 - manipulatory, trajektorie, IK

Sl. 20–21
Uczenie ze wzmocnieniem - PPO, SAC, Sim2Real

Sl. 22–23
Systemy wbudowane; STM32, RTOS, komunikacja CAN/I^2C

Sl. 24–25
Symulacja - Gazebo, Webots, NVIDIA Isaac Sim

Sl. 26–27
Bezpieczeństwo i standardy - ISO 10218, ISO 26262

Sl. 28–29
Studium przypadku - robot dostawczy i AV

Sl. 30
Trendy - Embodied AI, Foundation Models, ROS 3

Sala chwały

15 - K. Buszta
1 godzin 47 minut

14 - P. Szybiak
1 godzin 36 minut

13 - K. Bogdan
2 godzin 59 minut

12 - M. Bik
3 godzin 6 minut

11 - M. Snopkowski
3 godzin 26 minut

10 - A. Dulęba
3 godzin 45 minut

9 - J. Kucharski
3 godzin 51 minut

8 - M. Pokrywka
4 godzin 14 minut

7 - A. Cios
4 godzin 39 minut

6 - M. Kosciński
8 godzin 44 minut

5 - B. Sitek
10 godzin 20 minut

4 - M. Pereślucha
11 godzin 45 minut | 17:21 -> 14:28

3 - M. Hart
12 godzin 26 minut

2 - G. Solarz
13 godzin 21 minut

1 - B. Zacios
13 godzin 30 minut

Czym jest robot autonomiczny? Definicja i poziomy autonomii

"A robot is any automatically operated machine, that replaces human effort, though it may not resemble human beings in appearance or perform functions in a humanlike manner." - IEEE Standard 610.3

Trzy filary autonomicznego robota

Percepcja (Sensing)

Zbieranie danych o świecie: LiDAR, kamera, IMU, enkodery, dotyk. Kluczowe: jakość danych i kalibracja sensorów. Garbage in → garbage out.

Przetwarzanie (Processing)

Interpretacja sensorów → model świata → decyzja. Pipeline: percepcja → lokalizacja → planowanie → sterowanie. Często w czasie rzeczywistym.

Działanie (Actuation)

Silniki, serwomechanizmy, napędy, chwytaki. Pętla zwrotna: encoder → PID → PWM → silnik → enkoder. Dokładność ≈ pędnia napędowa.

Poziomy autonomii (SAE J3016 + robotics)

L0	**Brak autonomii** Człowiek wykonuje wszystko. Teleoperacja. Pilot drona z joystickiem.
L1	**Wspomaganie** Robot asystuje w jednym wymiarze. Serwonapęd stabilizujący kamerę.
L2	**Częściowa** Robot kontroluje kilka osi, człowiek nadzoruje. Cobot assist.
L3	**Warunkowa** Robot autonomiczny w określonych warunkach. Warehouse AMR.
L4	**Wysoka** Pełna autonomia w zdefiniowanym obszarze ODD. Delivery robot.
L5	**Pełna autonomia** Dowolne warunki, dowolne środowisko. Naukowy cel-nie osiągnięty.

Kinematyka robotów: Denavit-Hartenberg i Transformacje

Macierze transformacji i notacja DH

Python

```
# Parametry DH: a, d, alpha, theta
# a = długość członu (X-axis)
# d = przesunięcie (Z-axis)
# alpha = skręt osi Z (X-axis)
# theta = kąt obrotu (Z-axis)
import numpy as np
def dh_matrix(a, d, alpha, theta):
ct, st = np.cos(theta), np.sin(theta)
ca, sa = np.cos(alpha), np.sin(alpha)
return np.array([
[ct, -st*ca, st*sa, a*ct],
[st, ct*ca, -ct*sa, a*st],
[ 0, sa, ca, d],
[ 0, 0, 0, 1]])
# Forward kinematics: T_0_n = T01 @ T12 @ ... @ T(n-1)n
def forward_kinematics(dh_params, q):
T = np.eye(4)
for i,(a,d,alpha) in enumerate(dh_params):
T = T @ dh_matrix(a, d, alpha, q[i])
return T # pozycja i orientacja end-effector
```

Przestrzeń stawów vs. przestrzeń kartezjańska

$q \in \mathbb{R}^n$ (kąty/przemieszczenia stawów) $\leftrightarrow$ $x \in SE(3)$ (poza+orientacja). FK: q→x prosta. IK: x→q wieloznaczna lub niemożliwa.

Osobliwości

Konfiguracje gdy Jacobian J traci rząd → robot traci DOF. Np. wyciągnięte ramię. Wykrywanie: $\det(J \cdot J^T) < \varepsilon$ → alarm.

Kinematyka opisuje ruch robota geometrycznie, bez rozważania sił. Parametry DH to standard opisu łańcucha kinematycznego.

metody kinematyki odwrotnej (IK)

Analityczna (closed-form)

Dokładna, deterministyczna. Możliwa tylko dla robotów ≤6-DOF z uproszczonymi geometriami (np. PUMA 560). Wzory algebraiczne.

Jacobian Inverse / DLS

$J^+ = J^T(JJ^T)^{-1}$. Numeryczna iteracja $\Delta q = J^+ \cdot \Delta x$. DLS (Damped Least Squares) - odporny na singularności.

FABRIK

Forward And Backward Reaching Inverse Kinematics. Iteracyjny, szybki, naturalnie unika singularności. Standard w animacji i cobotach.

Optymalizacyjna (NLP)

IK jako minimalizacja $||FK(q)-x_{des}||^2$. Dodatkowe więzy: joint limits, kolizje, min torque. Używana w MoveIt 2 (TRAC-IK, KDL).

IK analityczna jest szybsza 100× od numerycznej, ale MoveIt 2 domyślnie używa KDL (numerycznej).
Dla cobotów real-time → TRAC-IK lub analityczna

Kinematyka robotów mobilnych: model i odometria

Napęd różnicowy (differential drive)

Python

```python
# Kinematyka napędu różnicowego
# v_L, v_R - prędkości lewego/prawego koła [m/s]
# L = rozstaw kół, R = promień koła

class DiffDriveKinematics:
    def __init__(self, wheel_radius, wheel_base):
        self.R = wheel_radius   # promień koła [m]
        self.L = wheel_base     # rozstaw kół [m]

    def forward(self, w_L, w_R):
        """omega [rad/s] → (v, omega_robot)"""
        v_L = w_L * self.R
        v_R = w_R * self.R
        v   = (v_R + v_L) / 2.0       # prędkość liniowa
        w   = (v_R - v_L) / self.L    # prędkość kątowa
        return v, w

    def odometry(self, x, y, theta, v, w, dt):
        """Całkowanie pozy po czasie dt"""
        theta_new = theta + w * dt
        x_new = x + v * np.cos(theta) * dt
        y_new = y + v * np.sin(theta) * dt
        return x_new, y_new, theta_new
```

Błędy odometrii

Poślizg kół, nierówności terenu, niedokładność enkoderów. Błąd akumuluje się O(√n). Korekcja: fuzja z LiDAR/IMU (EKF) lub korekcja pęta.

Holonomiczne vs. nieholonomiczne

Różnicowe i Ackermann: nieholonomiczne - nie mogą jechać bokiem. Mecanum/omnidirectional: holonomiczne - pełna swoboda w płaszczyźnie.

Roboty mobilne różnią się od manipulatorów. Nie mają sztywnego połączenia z bazą. Model kinematyczny + odometria = podstawa nawigacji.

Typy mobilnych kinematyk

Differential Drive

2 napędowe koła + 1–2 castery. Proste sterowanie, tanie. Turtlebot, Roomba. R_1≡prosto, R_2≡obrót w miejscu.

Ackermann (auto)

Przednie koła skrętne (jak auto). Minimalny promień skrętu. Lidar-based AGV, samochody autonomiczne.

Mecanum / Omni

Rolkowe koła 45°. Pełny ruch w płaszczyźnie XY + obrót. Wysokie koszty, wrażliwe na podłoże. Magazyny KUKA.

Tracked (gąsienice)

Wysokie tarcie i stabilność. Terenowe. Brak poślizgu. Stosowany w robotach ratowniczych i militarnych.

Legged (nóżkowy)

Spot (Boston Dynamics): 4 nogi, MPC+RL. Niezwykła lokomocja. Trudne sterowanie - aktywny obszar badań.

ROS 2: Robot Operating System 2, Architektura

ROS 2 to middleware dla robotów. Nie system operacyjny, lecz zbiór narzędzi, bibliotek i konwencji opartych o DDS (Data Distribution Service).

Aplikacja (węzły)	`rclpy/rclcpp · lifecycle nodes · composable nodes · executor · callback groups ·`
Middleware kliencka (rmw)	`rmw_fastrtps_cpp · rmw_cyclonedds · rmw_connextdds - wymienne przez env var`
DDS / RTPS Transport	`Fast-DDS (domyślny) · CycloneDDS · RTI Connext · QoS: reliability/durability/history`
Sieć / OS Layer	`UDP multicast (local) / TCP unicast (remote) · Linux / macOS / Windows · POSIX realtime`
Narzędzia	`colcon build · ros2 topic/service/action · rviz2 · rqt · ros2bag · foxglove studio`

ROS 2 vs ROS 1: brak Master node (discovery rozproszone przez DDS) · QoS reliability · nowe typy: Actions (preemptable) · native Windows/Mac · lifecycle nodes · Executors (multi-thread) · security (SROS 2).

ROS 2: Publisher, Subscriber, Serwis i Akcja (Python/C++)

Komunikacja w ROS 2: Topic (pub/sub, 1:N) · Service (req/resp, synchroniczny) · Action (długie zadanie z feedbackiem, cancelable).

Publisher & Subscriber (Python)

Python

```python
import rclpy
from rclpy.node import Node
from std_msgs.msg import Float32
from geometry_msgs.msg import Twist

class VelocityPublisher(Node):
    def __init__(self):
        super().__init__("vel_pub")
        self.pub = self.create_publisher(
            Twist, "/cmd_vel", 10)
        # QoS depth=10 (ostatnie 10 wiadomości)
        self.timer = self.create_timer(
            0.1, self.publish_vel)  # 10 Hz

    def publish_vel(self):
        msg = Twist()
        msg.linear.x  = 0.3  # [m/s]
        msg.angular.z = 0.1  # [rad/s]
        self.pub.publish(msg)
        self.get_logger().info(
            f"v={msg.linear.x:.2f}")

def main():
    rclpy.init()
    rclpy.spin(VelocityPublisher())
    rclpy.shutdown()
```

Action Server (nawigacja do celu)

Python

```python
from nav2_msgs.action import NavigateToPose
from rclpy.action import ActionClient
import rclpy; from rclpy.node import Node

class NavClient(Node):
    def __init__(self):
        super().__init__("nav_client")
        self._client = ActionClient(
            self, NavigateToPose,
            "navigate_to_pose")
    def send_goal(self, x, y, yaw=0.0):
        goal = NavigateToPose.Goal()
        goal.pose.header.frame_id = "map"
        goal.pose.pose.position.x = x
        goal.pose.pose.position.y = y
        self._client.wait_for_server()
        future = self._client.send_goal_async(
            goal,
            feedback_callback=self.feedback_cb)
        future.add_done_callback(self.goal_response)

    def feedback_cb(self, feedback):
        dist = feedback.feedback.distance_remaining
        self.get_logger().info(
            f"Dystans: {dist:.2f} m")
```

QoS Profile
RELIABLE+VOLATILE (domyślny) · BEST_EFFORT dla sensora · TRANSIENT_LOCAL+RELIABLE dla /map. Mismatch → brak połączenia.

ros2 launch
Startuj wiele węzłów: LaunchDescription + Node(). Grupy log, remapowania, parametry z YAML, composable containers.

Używaj lifecycle nodes dla krytycznych węzłów. gwarantuje poprawną inicjalizację i cleanup. Nav2 używa lifecycle zarządzanego przez lifecycle_manager.

Sensory robotów: LiDAR, Kamera, IMU i Enkodery

Wybór sensorów determinuje możliwości i koszty robota.
Nie istnieje jeden sensor do wszystkiego - fuzja multimodalna jest konieczna.

LiDAR 3D (VLP-16/Ouster OS1)

Time-of-flight: laser 905/1550nm, pomiar czasu lotu. Zasięg 100m, dokładność ±2cm. 16–128 linii skanujących. Chmura 700k pkt/s. Slabe: deszcz, mgła, lustrzane powierzchnie.

LiDAR 2D (RPLIDAR/Hokuyo)

Skanowanie 360° w jednej płaszczyźnie. Zasięg 12–25m, 0.33° rozdzielczość. Tani (RPLIDAR A3: 400$). Podstawa SLAM 2D - Turtlebot, AMR magazynowy.

Kamera stereo (ZED 2/RealSense)

Triangulacja: 2 kamery → mapa głębokości RGB-D. ZED 2: 20m depth, 120 FPS, SDK z VO. Intel D435: 10m, 90 FPS, VGA depth. IMU wbudowany.

IMU (VN-100, ICM-42688)

Accelerometr + żyroskop + (mag). 500–8000 Hz. Niezbędny dla EKF. VN-100: przemysłowy, EKF on-chip. Błąd integracji drift ~0.1°/s → potrzebna fuzja.

Enkodery (absolutne/inkrementalne)

Inkrementalny: impulsy na obrót. Absolutny: unikalna pozycja bez referencji (Grey code). AMT102: 8192 CPR, 7$. Kluczowe dla odometrii i PID.

Inne: ToF, sonar, dotyk, GPS

HC-SR04 ultrasonic: 2–400cm, ±3mm. ToF (VL53L5CX): macierz 8×8. GPS RTK: ±2cm (outdoor). Czujniki siły/momentu: FT300 (Robotiq) do cobot compliance.

Percepcja 3D: chmury punktów, PCL i PointNet

Chmura punktów to zbiór **P = {(x,y,z,i)ₙ}** - nieuporządkowany, setek tysięcy punktów per ramka. PCL (Point Cloud Library) to standard.

PCL pipeline - filtrowanie i segmentacja

```cpp
#include <pcl/point_cloud.h>
#include <pcl/filters/voxel_grid.h>
#include <pcl/segmentation/sac_segmentation.h>
#include <pcl/filters/extract_indices.h>

// Voxel grid downsampling (100k→5k pkt)
pcl::VoxelGrid<pcl::PointXYZ> vg;
vg.setInputCloud(cloud_raw);
vg.setLeafSize(0.05f, 0.05f, 0.05f); // 5cm voxel
vg.filter(*cloud_filtered);

// RANSAC plane segmentation (usuwanie podłogi)
pcl::SACSegmentation<pcl::PointXYZ> seg;
seg.setModelType(pcl::SACMODEL_PLANE);
seg.setMethodType(pcl::SAC_RANSAC);
seg.setDistanceThreshold(0.02); // 2cm
seg.setInputCloud(cloud_filtered);
pcl::ModelCoefficients::Ptr coeff(
    new pcl::ModelCoefficients);
pcl::PointIndices::Ptr inliers(
    new pcl::PointIndices);
seg.segment(*inliers, *coeff);
// coeff->values = [a,b,c,d] (równanie ax+by+cz+d=0)
```

Sieci neuronowe dla chmur punktów

PointNet (Qi et al., 2017)
Pierwsza sieć bezpośrednio na {(x,y,z)}: MLP per point → max pooling → global descriptor. Permutation invariant. 89.2% mAP ModelNet40. Podstawa wszystkich następnych.

PointNet++ (2017)
Hierarchiczne próbkowanie: farthest point sampling → local region → mini-PointNet. Lepsze na gęstych chmurach. Używany w autonomicznych samochodach.

VoxelNet / PointPillars (2018)
Voxelizacja → 3D CNN. PointPillars: voxele w kolumny 2D → 2D CNN → SSD detection head. 62 FPS na GPU. Waymo i Tesla używają podobnych architektur.

Open3D i ROS 2 pointcloud2
sensor_msgs/PointCloud2 w ROS 2. Open3D Python API: o3d.geometry.PointCloud → visualize, ICP registration, DBSCAN clustering. Interoperuje z NumPy.

Kalibruj LiDAR i kamerę wspólnie używając szachownicy 3D (ros2_camera_lidar_calibration). Bez kalibracji fuzja sensorów da błędne wyniki projekcji.

Computer Vision dla robotów: detekcja, segmentacja, głębokość

YOLO dla robotów: detekcja w czasie rzeczywistym

```python
from ultralytics import YOLO
import rclpy
from sensor_msgs.msg import Image
from cv_bridge import CvBridge

class YOLODetector(Node):
def __init__(self):
super().__init__("yolo_detector")
self.model = YOLO("yolov8n.pt")
# Dla robotyki: yolov8n < 3ms na Jetson
self.bridge = CvBridge()
self.sub = self.create_subscription(Image, "/camera/image_raw",
self.image_cb, 10)

def image_cb(self, msg):
img = self.bridge.imgmsg_to_cv2(msg)
results = self.model(img, conf=0.5)
for box in results[0].boxes:
# box.xyxy, box.cls, box.conf
self.process_detection(box)
```

Estymacja głębokości (monocular)

MiDaS / Depth Anything: relative depth z 1 kamery. ~25 FPS Jetson. Dokładność relatywna, wymaga kalibracji skalowej z LiDAR. Dobra dla planowania ścieżki.

Segmentacja semantyczna

SAM 2 (Meta) + YOLOv8-seg: real-time instance segmentation. DINOv2 jako backbone - zero-shot transfer na nowe klasy obiektów.

Kamera to najtańszy, najbogatszy informacyjnie sensor. Robot musi zamienić piksele w semantyczną wiedzę o środowisku.

Metody detekcji i ich cechy

YOLOv8 / YOLOv11

One-stage, anchor-free. 3–6ms GPU, 15ms Jetson AGX. COCO mAP 53. Najszybsze dla real-time robotyki. Model nano (3.2MB) na Raspberry Pi.

RT-DETR

Transformer-based, bez NMS. 54.8 mAP COCO. Wyższa dokładność niż YOLO. 9ms A100. Dobry gdy ważność > szybkość.

DINO / GroundedSAM

Open-vocabulary detection: 'find all bottles'. Zero-shot bez fine-tuningu. 5–10FPS - dla robotyki manipulacyjnej, nie nawigacji.

ArUco / AprilTag

Fiducial markers - deterministyczne, latencja < 1ms. Dokładność pozycji < 1mm przy kalibrowanej kamerze. Standard dla pick-and-place.

SLAM: Simultaneous Localization and Mapping

Problem SLAM formalnie

$$p(x_{1:t}, m \mid z_{1:t}, u_{1:t})$$

Dane: obserwacje $z_{1:t}$ (sensory), sterowania $u_{1:t}$.
Szukamy trajektoria robota $x_{1:t}$ + mapa m.

EKF-SLAM
Extended Kalman Filter. Stan = [robot_pose, landmarks]. O(n²) złożoność, nie skaluje się. Historia, ale ważny baseline.

GMapping / Gmapping (ROS)
Particle filter SLAM. 2D LiDAR. Dobry dla małych środowisk. Nadal używany
w Turtlebot3 demos.

Cartographer (Google)
Sparse pose graph + scan matching (CSM). 2D i 3D LiDAR. Submaps + loop closure. Przemysłowy standard dla AGV.

RTAB-Map
RGB-D + LiDAR SLAM. Loop closure z bag-of-words (visual). Eksportuje mapę 3D do Nav2. Bardzo kompletne rozwiązanie.

ORB-SLAM 3
Visual SLAM (kamera): ORB feature extraction → pose graph. Monocular/stereo/RGB-D. Stan akademickiego art. 2021.

SLAM rozwiązuje problem jajka i kury: zbuduj mapę żeby się zlokalizować, zlokalizuj się żeby budować mapę. Centralny problem robotyki mobilnej.

SLAM pipeline: kluczowe komponenty

Odometria (front-end)
Scan matching (ICP/NDT) lub visual odometry. Estymacja ruchu między klatkami. Szybkie (< 10ms), ale akumuluje błąd.

Loop Closure
Wykrycie wizyty w poprzednio odwiedzonej lokalizacji. Feature matching lub place recognition (NetVLAD, Scan Context). Koryguje drift.

Graph Optimization (back-end)
Pose graph: węzły=pozy, krawędzie=ograniczenia. Minimalizacja sumy kwadratów błędów. g2o, GTSAM, iSAM2. Rzadka macierz informacji.

Mapa
Occupancy grid (2D: 0/1/unknown), voxel map (OctoMap 3D), landmark map (punkty 3D), topologiczna (węzły+krawędzie).

Dla robotów indoor uruchom: ros2 launch slam_toolbox online_async_launch.py - gotowy SLAM 2D. Dla 3D outdoor: Cartographer z LiDAR 3D.

Planowanie ścieżki: A*, RRT*, Dijkstra i optymalizacja

implementacja i heurystyki A*

```python
import heapq

def a_star(grid, start, goal):
    # heurystyka: odległość Manhattan
    h = lambda n: abs(n[0]-goal[0])+abs(n[1]-goal[1])
    open_set = [(h(start), start)]
    g_score  = {start: 0}
    came_from = {}
    while open_set:
        _, cur = heapq.heappop(open_set)
        if cur == goal:
            path = []
            while cur in came_from:
                path.append(cur); cur=came_from[cur]
            return path[::-1]
        for dx,dy in [(-1,0),(1,0),(0,-1),(0,1)]:
            nb = (cur[0]+dx, cur[1]+dy)
            if grid[nb[0]][nb[1]] == 0:  # wolna komórka
                g = g_score[cur] + 1
                if g < g_score.get(nb, float('inf')):
                    g_score[nb] = g
                    came_from[nb] = cur
                    heapq.heappush(open_set, (g+h(nb),nb))
    return None  # brak ścieżki
```

Potencjalne pola (Potential Fields)

Cel = atraktor, przeszkody = repulsory. Szybkie, reaktywne. Wada: lokalne minima (robot utyka). Używane w local planner jako składowa.

Inflation layer - costmap

Przeszkody paddinowane buforem (robot_radius + margin). A* planuje z uwzględnieniem. NavStack costmap2d obsługuje automatycznie.

Planer ścieżki odpowiada na pytanie: jak dojść z punktu A do B unikając przeszkód? Dwa poziomy: global planner + local planner.

Algorytmy i ich porównanie

Dijkstra

A* z h=0. Gwarantuje optymalność. Wolniejszy niż A*. Podstawa dla wielu wariantów. O((V+E)log V).

A* (A-star)

Najszybszy kompletny dla grid map. h=Manhattan (4-conn) lub Chebyshev (8-conn). Admissible heuristic → optymalny.

D* Lite

Dynamic A* - replanning gdy zmienia się mapa. Świetny dla środowisk dynamicznych. Standard NASA Mars Rover.

RRT / RRT*

Szybkie drzewo losowych próbek. RRT*: asymptotycznie optymalny. Dobry dla wysokich DOF (6-DOF arm). Nie daje optymalnej ścieżki dla 2D nav.

Nav2 Smac Planner

A* na lattice / Hybrid-A* dla kinematic constraints (Ackermann). NavFn (Dijkstra-based). Oficjalny planer ROS 2.

Nav2: Navigation Stack ROS 2, architektura i konfiguracja

Nav2 to kompletny system nawigacji dla ROS 2. Zastępuje move_base z ROS 1. Modularny, lifecycle-managed, BehaviorTree-based.

BT Navigator

Orchestrator oparty o Behavior Tree (BehaviorTree.CPP). Koordynuje recovery behaviors, planner, controller.

Planner Server

Uruchamia global planner (NavFn/Smac). Wejście: /goal_pose → wyjście: /plan (ścieżka na mapie).

Controller Server

Local planner (DWB/MPPI/TEB). Śledzi globalną ścieżkę wysyłając /cmd_vel. Unika lokalnych przeszkód.

Costmap 2D

Global costmap (statyczna mapa + inflation) + Local costmap (sliding window 5×5m sensor data).

AMCL (Localization)

Adaptive Monte Carlo Localization - particle filter na gotowej mapie. 500–2000 cząstek. Wymaga map_server.

Recovery Behaviors

Spin · Backup · Clear costmap · Wait. Uruchamiane gdy robot utknął. Zdefiniowane w BT XML.

```
ros2 launch nav2_bringup navigation_launch.py  map:=/path/to/map.yaml  use_sim_time:=True
params_file:=nav2_params.yaml
ros2 topic pub /goal_pose geometry_msgs/PoseStamped '{header:{frame_id:"map"}, pose:{position:
{x:3.0,y:1.5}}}'
```

Sterowanie robotem: PID, regulatory i pętla zwrotna

Implementacja PID z anti-windup

Python

```
class PIDController:
    def __init__(self, Kp, Ki, Kd,
                 output_limits=(-1.0,1.0)):
        self.Kp, self.Ki, self.Kd = Kp, Ki, Kd
        self.integral = 0.0
        self.prev_error = 0.0
        self.limits = output_limits

    def compute(self, setpoint, measured, dt):
        error = setpoint - measured
        # Anti-windup: nie sumuj gdy na
limicie
        p_out = self.Kp * error
        d_out=self.Kd*(error-
self.prev_error)/dt
        # Sprawdź limity przed integrowaniem
        raw =
p_out+self.Ki*self.integral+d_out
        if self.limits[0] < raw <
self.limits[1]:
            self.integral += error * dt
        i_out = self.Ki * self.integral
        output = p_out + i_out + d_out
        self.prev_error = error
        # Clamp output
        return max(self.limits[0],
                   min(self.limits[1],
output))
```

PID (Proportional-Integral-Derivative) to najpowszechniejszy regulator w robotyce. Szacuje się, że 95% pętli sterowania w przemyśle to regulatory PID lub PI.

Metody strojenia PID

Ziegler-Nichols (Z-N)

Zwiększaj Kp do oscylacji → Ku, Tu. Oblicz: Kp=0.6Ku, Ti=0.5Tu, Td=0.125Tu. Szybka metoda - może być zbyt agresywna.

Tryb manualny + iteracyjny

Start: Kp mały, Ki=Kd=0. Zwiększaj Kp do oscylacji. Dodaj Ki dla steady-state error. Dodaj Kd dla tłumienia. Praktycznie najczęstszy.

Auto-tuning (relay feedback)

Wbudowany w sterowniki Beckhoff/Siemens. Układ przełączający (relay) → oscylacje kontrolowane → auto-oblicza Ku, Tu.

Cascade PID

Zewnętrzna pętla pozycji → setpoint wewnętrznej pętli prędkości → setpoint pętli prądu. Szybsze pętle wewnętrzne (1kHz current, 100Hz velocity).

Zaawansowane: LQR, MPC

Linear Quadratic Regulator, optymalne przy modelowaniu stanu. MPC (Model Predictive Control), predykcja N kroków. Standard w cobotach i AV.

```
# Zastosowanie: prędkość koła
```

Maszyny stanów i Behavior Trees, architektura decyzyjna

Implementacja Behavior Tree w Python

```python
import py_trees

# Liście (Behaviours)
class CheckBattery(py_trees.behaviour.Behaviour):
    def update(self):
        if battery_level > 20:
            return py_trees.common.Status.SUCCESS
        return py_trees.common.Status.FAILURE

class NavigateToGoal(py_trees.behaviour.Behaviour):
    def update(self):
        if nav_complete:
            return py_trees.common.Status.SUCCESS
        return py_trees.common.Status.RUNNING

# Drzewo: Sequence = AND, Selector = OR
root = py_trees.composites.Selector("Root", memory=False)

patrol = py_trees.composites.Sequence("Patrol",
memory=False)
patrol.add_children([
    CheckBattery(),
    NavigateToGoal(),
    PickObject()])

recharge = py_trees.composites.Sequence("Recharge",
memory=False)
recharge.add_children([GoToCharger(), Charge()])
root.add_children([patrol, recharge])
```

Decyzja: co robot ma teraz robić?
Maszyna stanów to dobry start, ale Behavior Trees (BT) skalują się lepiej dla złożonego zachowania.

Porównanie FSM vs Behavior Trees

Aspekt	FSM	Behavior Tree
Skalowalność	Eksplozja liczby stanów/tranzycji przy złożoności $O(n^2)$	Hierarchiczna, modułowa - dodaj poddrzewo bez zmian istniejącego
Czytelność	Diagram stanów czytelny do ~20 stanów	XML/graficzny, narzędzia (Groot2) - wizualizacja w runtime
Reaktywność	Trudna zmiana priorytetu - wymaga nowych tranzycji	Selector re-evaluuje każdy tick - naturalny priorytet
Współbieżność	Wymaga przebudowy na Statecharts	Parallel composite - wbudowana równoległość
Debugging	Logi tranzycji stanów	Groot2 live visualization, każdy node status kolorowany

Nav2 używa BehaviorTree.CPP v4 z gotowymi BT XML. Otwórz Groot2 IDE → Live Monitor podczas nawigacji, by zobaczyć aktywny węzeł drzewa w czasie rzeczywistym.

MoveIt 2: Planowanie Ruchu Manipulatorów

MoveIt 2 to de facto standard planowania ruchu dla ramion robotycznych w ROS 2. Integruje IK, kolizje, trajektorie i sterowanie.

MoveGroup Interface
Główne API. move_group.setPoseTarget() → plan() → execute(). Proste zadania w kilku linijkach Pythona/C++.

OMPL (planery)
Open Motion Planning Library: RRT*, PRM*, BIT*, LBTRRT. Każdy obsługuje 6-DOF w sekundach. Domyślny: RRT Connect.

FCL Collision Check
Flexible Collision Library: sphere/box/mesh primitives. Sprawdza self-collision i environment. < 1ms per check.

KDL / TRAC-IK
Kinematics solver. TRAC-IK 3× szybszy i bardziej niezawodny niż KDL (78% → 99% success rate). Zawsze używaj TRAC-IK.

Planowanie chwytania obiektów (pick and place)

Python

```python
# MoveIt 2 Python interface - pick & place
from moveit2 import MoveIt2
arm = MoveIt2(node, joint_names=["j1","j2","j3","j4","j5","j6"], base_link="base_link", end_effector="tool0",
group_name="manipulator")
# Zaplanuj do pozycji w przestrzeni kartezjańskiej
arm.set_pose_goal(position=[0.4, 0.2, 0.3], quat_xyzw=[0, 0.707, 0, 0.707])
success = arm.plan() # OMPL planning
if success: arm.execute() # wysyła do ros2_control

# Trajektoria kartezjańska (linearna, nie joint-space)
waypoints = [[0.4,0.2,0.3],[0.4,0.2,0.1],[0.4,0.3,0.1]]
arm.set_cartesian_path_goal(waypoints, max_step=0.005) # 5mm
success = arm.plan()        # Trajektoria kartezjańska
if success: arm.execute()   # wysyła do ros2_control
```

Uruchom MoveIt Setup Assistant (ros2 launch moveit_setup_assistant setup_assistant.launch.py) - generuje automatycznie SRDF, konfigurację collision matrix i launch files.

Uczenie ze wzmocnieniem: PPO, SAC i Sim2Real Transfer

RL dla robotów: zamiast programować zachowanie, robot uczy się go przez próby i błędy w symulacji. Sim2Real: przenieś nauczoną politykę na prawdziwy hardware.

PPO dla sterowania robotem: Stable-Baselines3

Python

```python
from stable_baselines3 import PPO
from stable_baselines3.common.env_util import (make_vec_env)
import gymnasium as gym

# Tworzenie środowiska (Gym / Gymnasium API)
env = make_vec_env("HalfCheetah-v4", n_envs=8)
# n_envs=8: równoległe środowiska → szybsze próbkowanie

model = PPO(
    "MlpPolicy",
    env,
    learning_rate=3e-4,
    n_steps=2048,    # kroków zbieranych przed update
    batch_size=64,
    n_epochs=10,
    gamma=0.99,      # discount factor
    gae_lambda=0.95, # GAE advantage estimation
    ent_coef=0.01,   # entropia → eksploracja
    verbose=1)

model.learn(total_timesteps=1_000_000)
model.save("ppo_cheetah")
# Inferencja: obs, _ = env.reset()
# action, _ = model.predict(obs, deterministic=True)
```

Algorytmy RL dla robotów

PPO (Proximal Policy Optimization)

On-policy. Stabilne, sample-efficient. Clip ratio ε=0.2. Złoty standard. Spot (BD) locomotion, OpenAI Dexterous Hand nauczyły się PPO.

SAC (Soft Actor-Critic)

Off-policy, maximum entropy RL. Automatyczny tuning temperatury. Bardziej sample-efficient niż PPO. Dobry dla ciągłych przestrzeni akcji (ramię).

Sim2Real Transfer - Domain Randomization

Problem: sim2real gap (tarcie, masa, opóźnienia różne). Rozwiązanie: randomizuj parametry w symulacji → polityka uodporniona na wariancję. NVIDIA używa Isaac Gym + DR.

Isaac Lab / IsaacGym

GPU-accelerated: 4096 środowisk równolegle na 1 GPU. 10× szybsze niż Gazebo. PyTorch natywnie. Robot learning state-of-art. AnyBotics ANYmal trenowany w Isaac.

Zacznij od Gymnasium MuJoCo envs (HalfCheetah, Ant, Humanoid) - stabilne benchmarki. Dopiero gdy polityka działa w symie, przenoś na hardware.

Systemy wbudowane: STM32, Raspberry Pi, Jetson i RTOS

Robot to hierarchia komputerów: mikrokontroler (< 1µs, PWM/SPI) → komputer pokładowy (nawigacja) → PC (planowanie, ML). Każdy poziom ma inne wymagania.

STM32F4/G4 (low-level ctrl)
Cortex-M4/M7, 168MHz. Sterowanie silnikami PWM, SPI do IMU, UART do czujników. Opóźnienie < 1µs. FreeRTOS lub bare-metal. micro-ROS → DDS agent.

Raspberry Pi 4/5 (mid-level)
ARM Cortex-A72, 4-core, 8GB RAM. ROS 2 natively. Brak gwarancji real-time (Linux). Dobry dla perception, planning. USB3 → kamera, WiFi, BT.

NVIDIA Jetson Orin (AGX/NX)
12 TOPS NVDLA + GPU 1792 CUDA + ARM A78. Dedykowany robot AI. TensorRT inference YOLOv8 < 3ms. Sterowanie robotem + ML na jednym modulu.

BeagleBone Blue / OpenCR
Specjalizowane do robotów: OpenCR (Turtlebot3) – STM32 + IMU + motor drivers na jednej płycie. BeagleBone: PRU (200MHz real-time coprocessors).

micro-ROS
ROS 2 na mikrokontrolerach (STM32/ESP32). micro-ROS Agent na RPi/PC. Topiki, serwisy przez XRCE-DDS (UDP/UART/CAN). STM32 + FreeRTOS oficjalnie wspierany.

CAN Bus w robotach
ISO 11898, 1Mbit/s. Odporny na zakłócenia. Multi-master. Sterowniki silników ESCON/ODrive/VESC komunikują przez CAN. ROS 2: ros2_socketcan.

ROS 2 real-time (Apex.OS)
Apex.OS = certyfikowana ASIL-D dystrybucja ROS 2 z gwarancjami real-time. Używana w autonomicznych wózkach widłowych STILL.

Real-Time Operating Systems

```c
/* FreeRTOS - sterowanie silnikiem, STM32 */
#include "FreeRTOS.h"
#include "task.h"

void motor_control_task(void *pvParam) {
TickType_t last_wake = xTaskGetTickCount();
for(;;) {
// Odczyt enkodera
int32_t pos = encoder_get_count();
// PID: setpoint → PWM
float pwm = pid_compute(
target_pos, pos, 0.001f); // dt=1ms
motor_set_pwm(pwm);
// Precyzyjne 1kHz (1ms period)
vTaskDelayUntil(&last_wake,
pdMS_TO_TICKS(1));
}
}
// Priorytet: motor_task(5) > nav_task(3) > log(1)
xTaskCreate(motor_control_task, "motor",
256, NULL, 5, NULL);
```

Komunikacja niskopoziomowa: CAN, UART, SPI, I²C, EtherCAT

Każdy protokół ma swoje zastosowanie. Dobry inżynier robotyki wybiera protokół pasujący do wymagań: przepustowość, dystans, real-time, koszty.

CAN Bus (ISO 11898)

1 Mbit/s 40m

Sterowniki silników (ODrive, VESC), IMU przemysłowy. Multi-master, priorytetyzacja. Samochody od 1986.

UART / RS-485

4 Mbit/s 1200m

GPS modemy, LiDAR (RPLIDAR UART), debug console. Punkt-punkt lub multi-drop (RS-485). Asynchroniczny.

SPI

50 Mbit/s 30cm

IMU (ICM-42688 10MHz SPI), Flash, enkodery absolutne. Full-duplex, synchroniczny. Master-slave.

I²C

3.4 Mbit/s 1m

Czujniki dystansu, ekspandery GPIO, OLED display. Tylko 2 przewody (SDA+SCL). Adresowanie 7-bit.

EtherCAT

100 Mbit/s 100m

Przemysłowe serwonapędy (Beckhoff AX5000). Jitter < 1µs. Każdy slave przetwarza ramkę w locie.

USB 3 / PCIe

5 Gbit/s 5m

Kamery RGB-D (ZED, RealSense), LiDAR 3D (Ouster). Wysoka przepustowość danych.

Symulacja: Gazebo Harmonic, Webots i NVIDIA Isaac Sim

Symulacja skraca czas developmentu 10×: testuj algorytmy bez ryzyka uszkodzenia hardware. Wierność symulacji to klucz do skutecznego Sim2Real.

Gazebo Harmonic (nowy Gazebo)

Następca Gazebo Classic. Zbudowany od zera: Ignition/gz-sim. Lepszy multi-robot, pluginy w Ignition Transport, OGRE 2 rendering. Integracja Nav2/MoveIt 2 przez ros_gz_bridge. Standard ROS 2.

Webots

Open-source, WSL-friendly. Prostszy start niż Gazebo. Wbudowane modele: TIAGo, Spot, UR10. Python supervisor API. Dobry dla dydaktyki.

CoppeliaSim (V-REP)

Profesjonalny, komercyjny. Wbudowana odwrotna kinematyka IKfast, siły kontaktu, przenośnik taśmowy. Zbliżony do rzeczywistości fizycznej.

NVIDIA Isaac Sim (Omniverse)

USD scene format. PhysX 5 physics. Fotorealistyczny rendering Lumen. 4096 równoległych środowisk GPU. Domain Randomization wbudowany. Przyszłość robotyki AI.

Gazebo - SDF i uruchamianie świata

Python

```python
# launch/simulation.launch.py
from launch import LaunchDescription
from launch_ros.actions import Node
from launch.actions import IncludeLaunchDescription

def generate_launch_description():
return LaunchDescription([
# Gazebo z customowym światem
IncludeLaunchDescription(
PythonLaunchDescriptionSource([
gz_sim_pkg, "/launch/gz_sim.launch.py"
]),
launch_arguments={
"gz_args": "my_world.sdf -r"
}.items()),
# Bridge ROS 2      Gazebo Topics
Node(package="ros_gz_bridge",
executable="parameter_bridge",
```

Modele URDF / SDF

URDF (XML): links, joints, visual, collision, inertia. xacro: makra i parametry. SDF: Gazebo native, więcej właściwości fizycznych. Eksport z Fusion 360 / SolidWorks.

ros2_control + Gazebo

hardware_interface::SystemInterface → Gazebo Plugin. Jeden kod kontrolera działa w symie i na hardware. Kluczowy wzorzec ROS 2.

Integracja: pełny stack nawigacyjny dla AMR

AMR (Autonomous Mobile Robot) w magazynie: cel = dotarcie do punktu, podniesienie towaru, powrót. Wymaga integracji wszystkich komponentów.

Pełny stack nawigacyjny -> przepływ danych:

Misja	Fleet Manager / REST API → cel (x,y,yaw) lub lista waypointów
Planowanie	Nav2 BT Navigator → Smac Planner (global) + DWB/MPPI (local) → /cmd_vel
Lokalizacja	AMCL na mapie OR Cartographer SLAM (dynamiczne środowisko) → /tf: map→odom→base_link
Percepcja	LiDAR 2D (RPLIDAR) → LaserScan → costmap. Kamera → YOLO → obstacle avoidance layer
Sterowanie	ros2_control: diff_drive_controller → PID PWM → silniki DC. Enkodery → odometria

Diagnoza: ros2 topic hz /scan (sprawdź 10Hz LiDAR) · ros2 topic echo /tf (sprawdź transformacje) · rviz2 → Add → TF · rqt_graph → sprawdź połączenia węzłów

Coboty i bezpieczeństwo: ISO 10218, Power and Force Limiting

Cobot (collaborative robot) = robot pracujący bezpiecznie obok człowieka bez fizycznej bariery. Wymaga certyfikacji i analizy ryzyka.

Tryby pracy cobota (ISO/TS 15066)

SSM - Speed & Separation Monitoring

Kamera/LiDAR monitoruje dystans człowiek-robot. Prędkość robota spada proporcjonalnie do zbliżenia. Zatrzymanie przy <100mm.

PFL - Power & Force Limiting

Czujniki siły/momentu w każdym stawie. Zatrzymanie gdy F>150N lub moment>80Nm. UR/KUKA/Fanuc cobots.

HGP - Hand Guiding Port

Człowiek prowadzi ramię ręcznie (gravity compensation). Teach-by-demonstration. FT sensor na flanszu.

STO - Safe Torque Off

Stop kategorii 0: natychmiastowe odcięcie napędu. Relay bezpieczeństwa + SIL 2. Zawsze jako ostateczność.

komunikacja i safety UR10e

```python
# UR10e przez ur_robot_driver (ROS 2)
# Prędkość max: 1m/s (safety-limited)
# Payload: 12.5kg, Reach: 1300mm

# joint_trajectory_controller → /joint_trajectory
from control_msgs.action import FollowJointTrajectory
from trajectory_msgs.msg import JointTrajectoryPoint

point = JointTrajectoryPoint()
point.positions = [0.0, -1.57, 1.57,-1.57, -1.57, 0.0]
# [base, shoulder, elbow, w1, w2, w3] rad
point.time_from_start.sec = 2  # 2 sekundy
# Safety: tool_contact_threshold = 50N
# Jeśli FT sensor przekroczy → E-STOP
# Konfiguracja w PolyScope Safety Panel
```

Analiza ryzyka (ISO 12100)

Identyfikacja zagrożeń → ocena ryzyka (Severity × Probability × Avoidability) → redukcja przez design → ostrzeżenia → PPE. Dokumentacja obowiązkowa dla CE.

Popularne coboty 2024

UR10e (Universal Robots) · KUKA LBR iisy · Fanuc CRX · ABB GoFa/Swifti · Franka Emika Panda (research standard) · Techman TM14.

NEVER connect cobot directly to ROS 2 without safety controller in the loop. Używaj certified safety PLC (e.g. Pilz PNOZ) jako hardware guard.

Systemy wielorobotowe: koordynacja, rój i fleet management

Wiele robotów może robić więcej niż jeden - ale koordynacja to nowy problem: unikanie kolizji, rozdzielanie zadań, komunikacja.

Multi-robot w ROS 2 (namespace)

```python
# Każdy robot to osobna przestrzeń nazw
# robot_1/cmd_vel, robot_2/cmd_vel
# robot_1/scan, robot_2/scan
# launch/multi_robot.launch.py
from graphene import Node

robots = [
    {"name":"robot_1","x":0.0,"y":0.0},
    {"name":"robot_2","x":2.0,"y":0.0},
    {"name":"robot_3","x":4.0,"y":0.0}]
for robot in robots:
    ld.add_action(Node(
        package="nav2_bringup",
        executable="bringup_launch.py",
        namespace=robot["name"],
        parameters=[{
            "use_namespace": True,
            "robot_name": robot["name"],
        }]))

# MAPF: Multi-Agent Path Finding
# CBS (Conflict-Based Search) dla optymalności
```

MAPF - planowanie wielu agentów

Conflict-Based Search (CBS): dekompozycja na single-agent + tree konfliktów. Suboptymalne: WHCA* (windowed). Amazon Robotics: 750k robotów w magazynach.

Fleet Management System (FMS)

Task allocation (aukcja, Hungarian method). Traffic management (virtual lanes, traffic lights). Open-source: OpenRMF. Komercyjny: MiR Fleet, OTTO Fleet.

Robotics roju (Swarm Robotics)

Flocking (Craig Reynolds Boids)

Trzy reguły: Separation (unikaj sąsiadów) + Alignment (wyrównaj prędkość) + Cohesion (trzymaj się grupy). Emergentne zachowanie z prostych reguł lokalnych.

Stigmergy - komunikacja przez środowisko

Jak mrówki: zostawiaj ślad (virtual pheromone). Robot czyta/pisze do shared map. Nie wymaga bezpośredniej komunikacji. Skalowalny do tysięcy agentów.

Kilobot / e-puck / Crazyflie

Kilobot (Harvard): 3cm, IR komunikacja, 1024 roju. e-puck 2: edukacyjny. Crazyflie 2.1: nano-dron, swarm flights z Crazyswarm2 ROS 2 package.

Przeszkody: komunikacja i skalowanie

Bandwidth: N^2 wiadomości dla all-to-all. Rozwiązanie: broadcast lokal. Konsensus (Raft/Paxos) przy centralizowanym FMS. ROS 2 DDS ograniczone do ~50 robotów bez tuningu.

Amazon Robotics / Kiva

750k robotów Proteus w centrach dystrybucji. Siatka 30cm - roboty przywożą półki do człowieka. 3× szybszy fulfillment. $775M akwizycja Kiva 2012.

Studium przypadku: autonomiczny samochód (SAE L4)

AV (Autonomous Vehicle) to najbardziej wymagające wdrożenie robotyki: pełne bezpieczeństwo życia, każdy edge case musi być obsłużony.

Percepcja

LiDAR 128-line (Luminar Iris) + kamera 360° (8×) + radar 4D (Continental ARS540). Sensor fusion: BEV (Bird's Eye View) representation.

Predykcja

Przewidywanie zachowania pieszych/aut na 5s. Transformer-based (Wayformer, MTR). Multiple hypotheses z probability distribution.

Planowanie

Structured prediction: reguły drogowe + optymalizacja trajektorii. MPC z constraint satisfaction. Waymo używa Monte Carlo rollouts.

Sterowanie

Stanley / Pure Pursuit + MPC dla śledzenia ścieżki. Lat/Lon control: kierownica + gaz/hamulec. Aktuatory by-wire (EPS + EBS).

Sprzęt obliczeniowy - NVIDIA DRIVE Thor / Orin

DRIVE Orin: 254 TOPS. DRIVE Thor (2025): 2000 TOPS. Redundancja 2+1: primary compute + safety monitor + lockstep. Fail-operational: 3s maneuver po awarii compute.

Porównanie 2024: Waymo One (L4, 25 miast USA) · Baidu Apollo Go · WeRide · Tesla FSD Beta (L2+, ADAS). Waymo: 0 fatalities / 10M km. Ludzie: 1.37 deaths / 100M km.

Apollo Open Platform: Baidu open-source AV stack. GitHub: ApolloAuto/apollo. 100+ modułów, własny simulator (Dreamview). Używany przez 200+ firm.

Robot dostawczy (studium przypadku sl. 29) jest lepszym projektem niż AV - niższe prędkości (<20km/h), mniejsze ODD, bardziej osiągalny jako projekt studencki.

Standardy bezpieczeństwa: ISO 10218, ISO 26262, IEC 61508

Bezpieczeństwo robotów to wymóg prawny (CE marking, OSHA) i etyczny. Systemy autonomiczne muszą dowieść bezpieczeństwa formalnie.

Hierarchia standardów bezpieczeństwa

IEC 61508	Elektryczne/elektroniczne systemy safety-critical	SIL 1–4	Standard nadrzędny. Definiuje SIL (Safety Integrity Level). Podstawa dla ISO 26262 i ISO 10218.
ISO 10218-1/2	Roboty przemysłowe i komórki robotyczne	PLd/SIL2	Część 1: robot. Część 2: instalacja. Definiuje PL (Performance Level) a-e według EN ISO 13849-1.
ISO/TS 15066	Roboty współpracujące (coboty)	PLd/SIL2	Uszczegółowia 10218 dla cobotów: SSM, PFL, HGP tryby. BioMechanical Limit Tables dla sił zderzenia.
ISO 26262	Automotive ASIL (pojazdy drogowe)	ASIL A–D	ASIL D = najwyższy. Dotyczy AV, systemów ADAS. Deterministic failure rate 10^{-8}/h dla ASIL D.
UL 4600 (AV)	Pojazdy autonomiczne - brak kierowcy	Safety Case	Standard oparty o Safety Case (argumentacja). AWS, Waymo pracują z UL 4600. USA/CA AV regulations.

Dla projektu studenckiego: przeprowadź uproszczoną analizę ryzyka (FMEA: co może pójść źle → skutek → zabezpieczenie). Minimum: E-STOP sprzętowy + ogranicznik prędkości.

Studium przypadku: autonomiczny robot dostawczy

Starship Technologies, Nuro, Amazon Scout: roboty trotuar-level delivery.
SAE L4 w ograniczonym ODD (trotuary, maks 20km/h, geofence).

Architektura Starship (uproszczona)

Sensory
9 kamer + LiDAR 2D + ultrasonic + GPS RTK. Kamera 360° główna. GPS RTK ±2cm dla fine localization przy skrzyżowaniach.

Mapa
Wstępna mapa: HD survey vehicle. Update: każdy robot aktualizuje shared cloud map. Waypoints to trotuary, miejsca dostawy.

Percepcja
Detekcja pieszych: kamera + depth estimation. Bariera 50cm → zatrzymanie. Cloud-assisted dla edge cases (video streaming).

Nawigacja
A* po graphie trotuarowym (pre-mapped). Local: DWA. Skrzyżowanie: oczekiwanie na zielone (detekcja sygnalizacji YOLO).

Fleet
1000+ robotów w 100+ miastach. Centrala widzi wszystkie roboty real-time. Teleoperator interweniuje dla < 1% dostaw.

Kluczowe metryki i wyzwania

Python

```python
# Specyfikacja techniczna (Starship Gen 3)
robot_specs = {
    "masa_kg"      : 50,          # + 10kg ładunek
    "max_speed_kmh": 6,           # trotuar-safe
    "bateria_Wh"   : 450,         # ~6h / 20km range
    "sensory"      : "9 cam + LiDAR + USS + GPS RTK",
    "compute"      : "NVIDIA Jetson Orin NX",
    "lockbox"      : "code unlock via mobile app",
    # KPI produkcyjne:
    "delivery_success": 0.998,    # 99.8% kompletnych
    "teleop_rate"     : 0.008,    # 0.8% z interwencją
    "mtbf_hours"      : 500,      # mean time between failure
}
```

Stairs & kerbs
Nie może wchodzić po schodach - musi omijać. HD mapa: predefiniowane przejścia dla pieszych z ramą.

Agresywne dzieci
Rzeczywisty problem! Monitoring kamery + remote intervention. Stalowe obudowanie kół. Brak danych osobowych.

Pogoda
Deszcz i śnieg degradują kamery. IP65 obudowa. LiDAR odporny. Grzałki przednia kamera / koła w opcji.

Regulacje
Różne przepisy miasto/kraj. UK: Micromobility law. USA: state-by-state. EU: NIS2 dla connected robots.

Projekt studencki: zbuduj mini delivery robot na ROS 2 + Nav2. Hardware: Turtlebot4 (€1200) lub samodzielnie RPi4 + RPLIDAR + DC motors + enkodery.

Trendy: embodied AI, Foundation Models i ROS 3

Rewolucja 2023–2025: Large Language Models + robotyka = roboty rozumiejące polecenia naturalne. Google, OpenAI, Tesla wchodzą w robotykę.

RT-2 / RT-X (Google DeepMind)

Robotics Transformer: VLM (Visual-Language Model) end-to-end. Input: obraz + tekst polecenia. Output: akcje robota. Zero-shot transfer między robotami. Open X-Embodiment dataset 1M epizodów.

π_0 - Physical Intelligence

Foundation model dla humanoidów. Pre-trained na danych fizycznych → fine-tuned na task. Figure 01 / Boston Dynamics Atlas z LLM. 2025: pierwsze produktywne humanoidale w fabrykach.

Diffusion Policy / ACT

Imitation learning z demonstracji ludzkich: 30min demo → polityka robota. ACT: Action Chunking Transformer (Tony Zhao, Stanford). Mobile ALOHA: sprzątanie, gotowanie.

OpenVLA / Octo (open-source)

Open-source foundation models dla robotów. Octo (UC Berkeley): 800k demonstrations, fine-tunable. OpenVLA 7B: pełna waga dostępna. Jak HuggingFace dla robotów.

ROS 3 / ROS 2 Iron+

ROS 2 Iron (LTS 2023): stable. Jazzy (2024). Kilpatrick (2026). ROS 3 w dyskusji: Rust-based communication, unified IDL, native TypeScript. micro-ROS 2.0 dla MCU.

Humanoidale (2025–2030)

Figure 01/02 (Figure AI) · Optimus Gen 2 (Tesla) · Atlas (BD) · Unitree H1. Cena target: $20k (jak samochód). BYD używa Unitree G1 w fabrykach. AGI w ciele fizycznym.

Testowanie i diagnostyka: metryki, rviz2 i rosbag2

Dobry robot to testowany robot. Zdefiniuj metryki przed implementacją - nie po.
Nagrywaj dane do późniejszej analizy.

Kluczowe narzędzia diagnostyczne

rviz2
Wizualizacja: TF tree, costmap, LaserScan, plan, path, markers. Niezbędny do debugowania percepcji i nawigacji.

rqt_graph
Graf węzłów i połączeń. Widoczne publisher/subscriber. Szybka diagnoza: czy topic istnieje? Kto publikuje?

ros2bag2
Nagrywaj wszystkie topiki: ros2 bag record -a. Replay: ros2 bag play. Analiza post-mortem awarii i benchmarking.

PlotJuggler
Real-time plot topików numerycznych. Lepszy niż rqt_plot. CSV eksport. Niezbędny dla strojenia PID i analizy trajektorii.

Foxglove Studio
Nowoczesny rviz2 z przeglądarki. WebSocket bridge. Najlepszy UX dla prezentacji i remote debugging.

Metryki ewaluacji robota nawigacyjnego

Python

```python
# Skrypt ewaluacji nawigacji
import rclpy, yaml
from nav2_simple_commander.robot_navigator import (
    BasicNavigator, TaskResult)
def evaluate_navigation(waypoints, n_trials=10):
    nav = BasicNavigator()
    results = {"success":0,"collisions":0,"times":
[],"path_lengths":[]}
    for trial in range(n_trials):
        for wp in waypoints:
            start = time.time(); nav.goToPose(wp)
            while not nav.isTaskComplete(): time.sleep(0.1)
            result = nav.getResult()
            elapsed = time.time() - start
            if result == TaskResult.SUCCEEDED:
                results["success"] += 1
                results["times"].append(elapsed)
            else:
                results["collisions"] += 1
    # KPI
    sr = results["success"]/(n_trials*len(waypoints))
    print(f"Success Rate: {sr:.1%}")
    print(f"Avg Time: {np.mean(results['times']):.1f}s")
    return results
```

Metryki nawigacji
Success Rate (SR) · Average Time To Goal (ATTG) · Average Path Length (APL) · Path Length Ratio (PLR = actual/optimal) · Collision Rate per km

Złota zasada: zawsze miej E-STOP w ręku podczas testów na hardware. Zanim puścisz robota, sprawdź: max_vel < 0.5m/s, costmap inflation_radius > robot_radius.

Przyszłość autonomicznych robotów

Embodied AI

LLM + Vision + robot = instrukcje językiem naturalnym. RT-2, OpenVLA, π_0. Zero-shot task transfer.

Humanoidale

Figure 02, Tesla Optimus, Atlas - do fabryki 2025–2027. Cena docelowa <$20k. AGI w ciele fizycznym.

Swarm & Fleet

1M+ robotów Amazon, DHL. Open-source Fleet: OpenRMF. ROS 2 wspiera multi-robot out-of-box.

Sim2Real (GPU)

Isaac Lab: 4096 środowisk/GPU. Domain randomization → zero-shot transfer. PPO polityka w 1h treningu.

Safety & Trust

UL 4600, ISO 26262 ASIL-D. Formal verification (TLA+, Coq). Explainable AI dla robotów bezpiecznych.

ROS 2 → Future

Jazzy LTS stabilne. ROS 3 w planowaniu (Rust-based). micro-ROS 2.0 STM32. ROS-Industrial standard.

Kluczowe wnioski

- ROS 2 + Nav2 + MoveIt 2 to produkcyjny ekosystem - opanuj go, a skalujesz na dowolnego robota
- SLAM i planowanie ścieżki to rozwiązane problemy - kluczowe jest dobra inżynieria integracji i kalibracja
- Bezpieczeństwo nie jest opcją - E-STOP, analiza ryzyka i standardy ISO są częścią projektu, nie dodatkiem
- Sim2Real: zawsze testuj w symulacji najpierw. Gazebo + Isaac Lab skracają development 10×

Koniec