INSTITUTO UNIVERSITÁRIO DE LISBOA

FINAL PROJECT

AUTONOMOUS WAREHOUSE INSPECTION ROBOT

TOPICS IN INTELLIGENT ROBOTICS

António Cruz (140129)

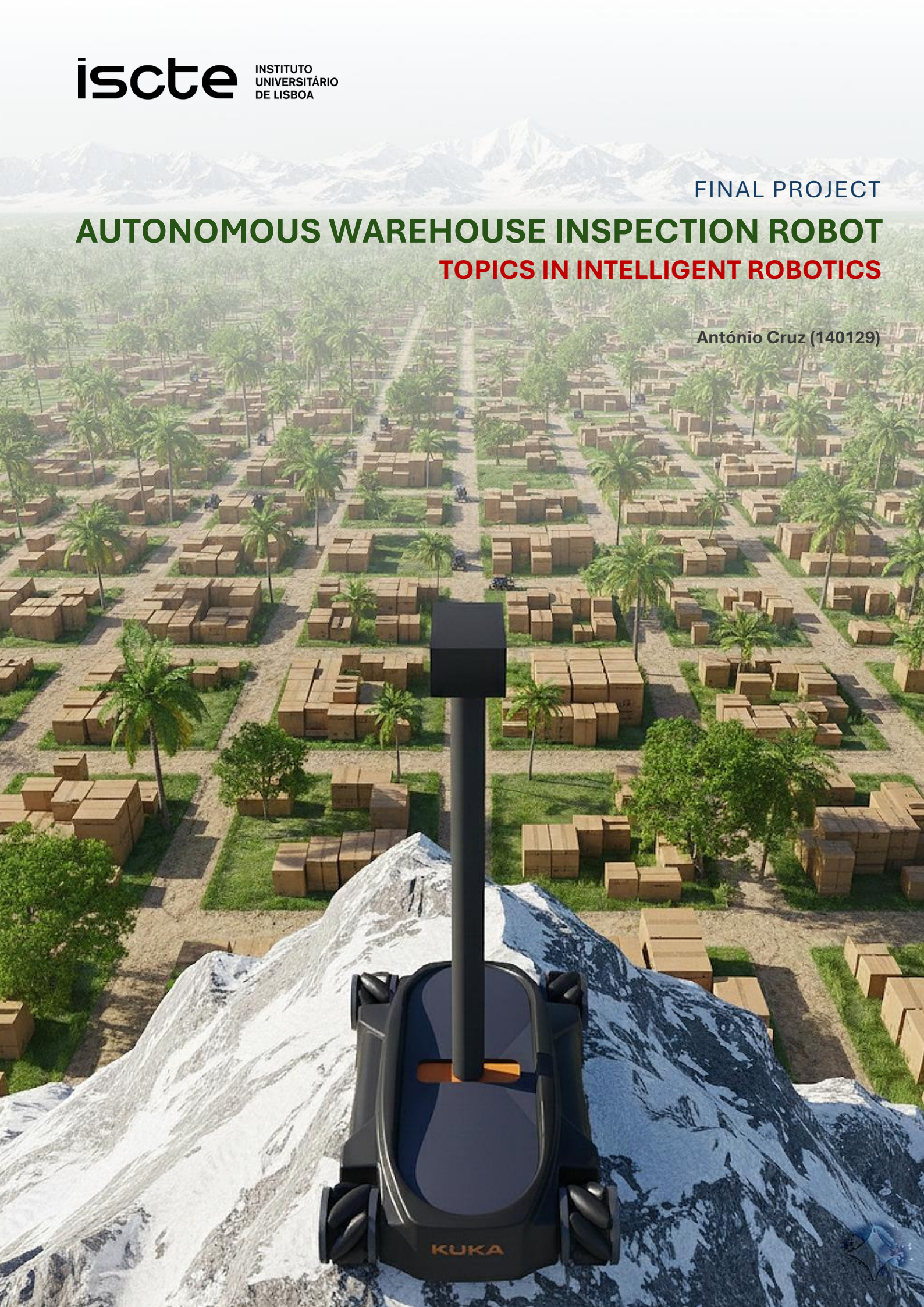

Table of Contents

1. Introduction and Motivation

1.1 Context

Warehouse logistics plays a central role in modern supply chains, where inventory management and quality control represent a significant share of operational costs. In large-scale storage facilities, goods are typically organized across multiple aisles and shelving units, with hundreds or thousands of individual packages requiring periodic inspection. Damaged packaging can result in product losses, customer complaints, and safety hazards, making early detection a priority for warehouse operators.

Manual inspection of stored goods remains the prevailing practice in many facilities. Human inspectors walk through aisles, visually assess each package, and record their findings. This process is time-consuming, labor-intensive, and inherently prone to inconsistency - fatigue, oversight, and subjective judgment all contribute to missed defects or false positives. As warehouses grow in scale and throughput demands increase, the limitations of manual inspection become more pronounced.

Autonomous mobile robots offer a compelling alternative. Equipped with cameras and onboard intelligence, a robot can systematically traverse a warehouse, capture images of every stored item, and classify their condition without human intervention. When combined with machine learning, such a system can not only match human inspection accuracy but also operate continuously, consistently, and at scale.

1.2 Project Goals

This project develops an autonomous warehouse inspection system built around a mobile robot operating within the Webots robotics simulator. Rather than focusing solely on the robot itself, the project delivers an integrated system that spans simulation, perception, classification, and human interaction. The core objectives are:

1. Design and implement an autonomous mobile robot capable of navigating a warehouse environment and systematically inspecting every stored box, capturing images for quality assessment.
2. Build a procedural world generation pipeline that creates random warehouse layouts with configurable damage types, enabling the automated production of large, labeled image datasets for machine learning.
3. Train a custom convolutional neural network to classify box conditions - damaged, undamaged, or empty position - using exclusively synthetic data captured by the robot during automated patrol runs.
4. Develop a web-based control interface that decouples the simulation from the Webots desktop application, enabling remote monitoring, voice-based robot interaction through natural language, and real-time inventory tracking with ground truth comparison.

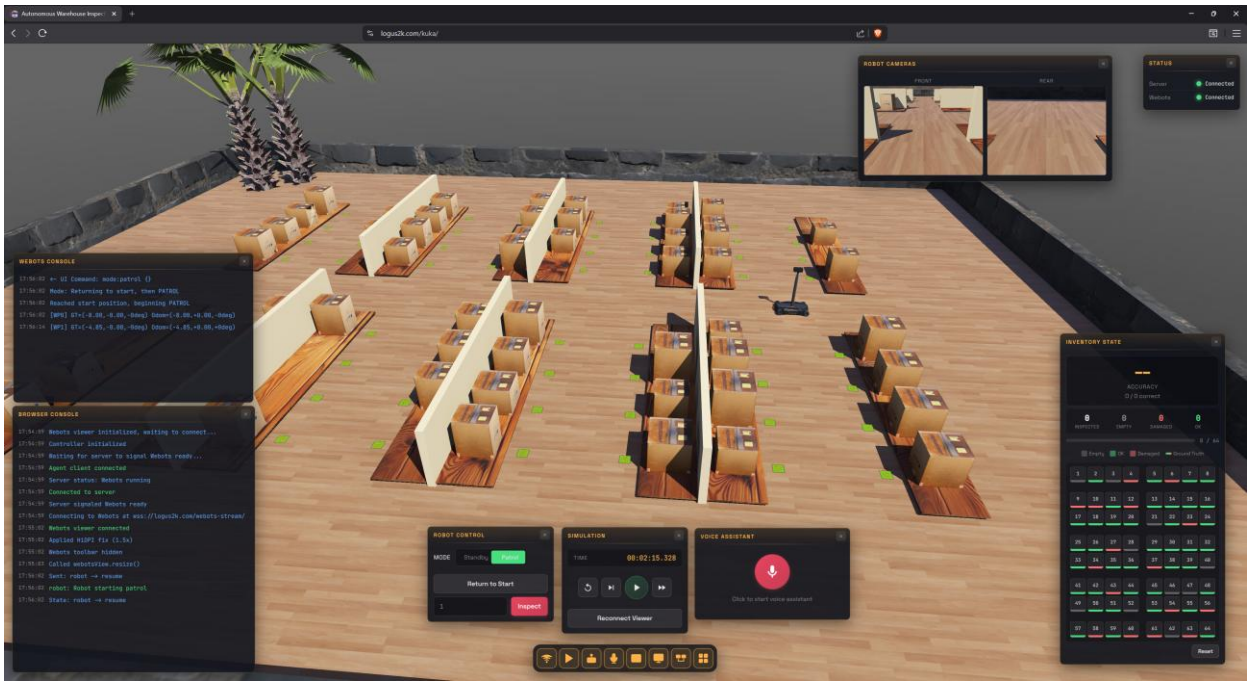

Figure 1.1 *- Overview of the warehouse inspection system running in the browser from a self-hosted domain, showing the 3D simulation viewer, dual camera feeds, inventory panel, and voice control interface.*

1.3 Motivation

A defining feature of this project is that the warehouse environment is never fixed. The entire world (shelf placement, box positioning, damage types and severities, empty positions) is regenerated procedurally on every launch from a set of numeric parameters in a JSON configuration file. This transforms each simulation run into an independent experiment and carries several advantages that shaped the development workflow.

Procedural regeneration provides automatic ground truth labeling. Every box placed in the generated world has a known condition recorded at generation time, flowing directly into the training pipeline as image labels and into the runtime accuracy tracker as the reference for scoring predictions. No manual annotation is required at any stage.

Regeneration also enables dataset scaling through automation. By running the simulation in a loop - regenerating the world, executing a full patrol, and collecting all 64 images per run - the system accumulated 16,798 labeled training images. Because every run produces a unique warehouse, no two images share the same box geometry, damage pattern, or shelf arrangement, which is the property needed to train a model that generalizes rather than memorizes.

Finally, every evaluation run tests the trained model against a warehouse it has never seen, providing built-in cross-validation without a manually curated test set.

2. Scenario and Webots World Description

2.1 Warehouse Environment

The simulation reproduces a rectangular warehouse floor measuring 20 by 15 meters, enclosed by low perimeter walls. The interior is organized into four parallel storage aisles, each flanked by shelving units that hold the boxes to be inspected. The layout follows the conventions of a typical small-to-medium warehouse, with enough complexity to exercise navigation, image capture, and classification while remaining tractable for systematic evaluation.

The warehouse floor uses Webots coordinate conventions, with the X axis spanning the width, the Y axis spanning the depth, and the Z axis pointing upward. The robot operates on the ground plane at Z = 0, and boxes rest on shelves slightly above the floor.

Parameter	Value	Description
Arena width	20 m	Total floor extent along X
Arena depth	15 m	Total floor extent along Y
Wall height	0.5 m	Perimeter wall height
Number of aisles	4	Parallel storage aisles
Shelf length	3.0 m	Length of each shelf unit along Y
Shelf spacing	1.5 m	Gap between consecutive shelf units
Aisle width	2.5 m	Clear passage between opposing shelves

2.2 Procedural World Generation

A central design decision is that the warehouse world is never hand authored. Instead, a Python script (generate_world.py) reads a JSON configuration file (config.json) and produces the complete simulation environment from scratch on every launch. This procedural approach has three outputs: a VRML-based .wbt world file that Webots loads directly, a patrol_waypoints.json file that defines the robot's navigation plan, and the internal metadata that records the ground truth condition of every box.

The generator operates in two passes. In the first pass, it computes the physical layout - shelf positions, shelf depths (narrower for wall-adjacent units, wider for back-to-back inner shelves), and the precise coordinates where boxes will be placed. In the second pass, it populates those positions with boxes, applying randomized damage and leaving some positions empty according to configurable ratios. Because both the damage distribution and the empty positions are randomized, every world launch produces a unique scenario. This property is essential for two purposes: it prevents the classification model from memorizing specific layouts during training, and it ensures that every evaluation runs tests against a genuinely novel configuration.

The configuration file controls all significant parameters - arena dimensions, number of aisles, boxes per aisle, damage ratios, damage severity, and visual options such as floor labels and grid overlays. Changing a single value and relaunching the simulation is sufficient to produce a different warehouse layout.

Figure 2.1 *- Top-down view of the procedurally generated warehouse environment, showing the four aisles with shelving units and box placements.*

2.3 Box Placement and Numbering

Each aisle has boxes on both its left and right sides, placed on shelves at known Y positions. With four boxes per side per shelf unit and two shelf units per aisle, the default configuration produces eight box positions per side, for a total of 64 positions across all four aisles. Not all positions are occupied: a configurable empty ratio (default 15%) leaves some slots vacant to simulate real warehouse conditions where inventory levels vary.

Boxes are numbered sequentially in column-major order - that is, all positions along one side of one aisle are numbered before moving to the next side. Aisle 0's left side holds positions 1 through 8, its right side holds 9 through 16, aisle 1's left side holds 17 through 24, and so on through position 64 on the right side of aisle 3. This numbering convention is consistent across the waypoint file, the robot controller, and the inventory tracking interface, ensuring that a box identified as "box_23" refers to the same physical location throughout the entire system.

Each box position is recorded with its 3D coordinates, aisle index, aisle side (left or right), and ground truth condition - damaged, undamaged, or empty. This metadata serves as the reference against which the classification model's predictions are later compared.

2.4 Procedural Damage Generation

One of the more technically involved aspects of the project is the procedural generation of realistic box damage. Rather than using a library of pre-modelled damaged meshes, the system synthesizes damage algorithmically by manipulating the box geometry at the vertex and triangle level. This approach produces an effectively unlimited variety of damage

patterns, which is critical for training a classification model that generalizes rather than memorizing specific examples.

Geometry Foundation

Each box is represented as six independent face geometries - front, back, left, right, top, and bottom - each consisting of an exterior surface, an interior surface (offset inward by a configurable wall thickness), and associated UV texture coordinates. The FaceGeometry class manages the vertices, triangles, and UV mappings for each face, and provides methods for adding new vertices, clearing and rebuilding triangle lists, and querying face dimensions and orientation. All damage operations work by modifying these face geometries in place: adding new vertices, retesselating the affected face, and creating edge surfaces where material has been removed.

Damage Types

The system implements four categories of damage, each with its own generation algorithm and configurable parameters.

Holes are irregular polygonal punctures through a box face. The algorithm generates a random polygon in UV space by distributing vertices around a center point at varying radii and angles, controlled by an irregularity parameter that ranges from 0 (regular polygon) to 1 (highly jagged). The polygon is then projected onto the 3D face surface and used to retesselate the face with the hole region removed. Both the exterior and interior surfaces are rebuilt, and edge triangles are generated around the hole perimeter to render the wall thickness visible through the opening. An overlap detection routine ensures that multiple holes on the same face do not intersect.

***Figure 2.2** - A procedurally generated hole on a box face. The irregular polygon shape and visible wall thickness through the opening contribute to visual realism.*

Dents are inward deformations that simulate impact damage. The algorithm creates an irregular rim polygon on the face surface, and a center vertex displaced inward along the face normal by a configurable depth. The face is then retesselated as a pyramid-like depression: the outer region connects the face corners to the rim vertices, and the inner region fans from the rim to the depressed center. The result is a concave region that catches light differently from the surrounding flat surface, making it visually distinguishable.

Corner tears simulate damage at box corners where the cardboard has been torn away. The algorithm generates an irregular boundary path from one edge of the face to an adjacent

edge, passing through the corner region. The portion of the face beyond this boundary is removed, and edge triangles are created along the tear line to expose the wall thickness. The tear shape is controlled by a size parameter (as a fraction of face dimensions) and an irregularity parameter that introduces random displacement along the boundary path.

Edge tears follow a similar principle but originate from the middle of a face edge rather than a corner. The algorithm creates a parabolic incursion into the face, with the maximum depth at the midpoint of the tear and tapering to zero at both ends where the tear meets the edge. This produces a notch-like defect with an irregular boundary.

Severity Scaling and Distribution Control

All damage parameters - sizes, depths, vertex counts, and probabilities - are scaled by a severity system with three levels: light, moderate, and severe. Each level applies multipliers to the base configuration values, producing proportionally larger or smaller defects. The severity can be set globally or randomized per box.

The damage distribution is also configurable. In the default mode (front_required), the face of the box that is visible from the aisle always receives damage, with a 50% probability of additional damage on the top face. This ensures that damage is always potentially detectable by the robot's cameras, which is important for both dataset quality and classification evaluation. Alternative distribution modes include top_only, front_only, both, and fully random.

Aisle-Aware Damage Placement

The system considers which side of the aisle a box occupies when determining which face constitutes the "front" - that is, the face visible to the robot's camera. For boxes on the left side of an aisle, the front face points toward the positive X direction; for boxes on the right side, it points toward negative X. If the box has a random Z rotation applied, the system computes the dot product of each side face's rotated normal with the camera view direction to identify which face is visible. This ensures that damage is placed where the camera can see it, regardless of the box's orientation.

VRML Output

Each damaged box is serialized as a Webots Solid node containing six Shape children, one per face. Each shape uses an IndexedFaceSet geometry with the modified vertex coordinates, triangle indices, and UV mappings, textured with the built-in Webots Cardboard appearance. The box also carries a description field encoding its damage state ("damaged:true" or "damaged:false"), which serves as the ground truth label accessible to the robot controller through the Webots Supervisor API.

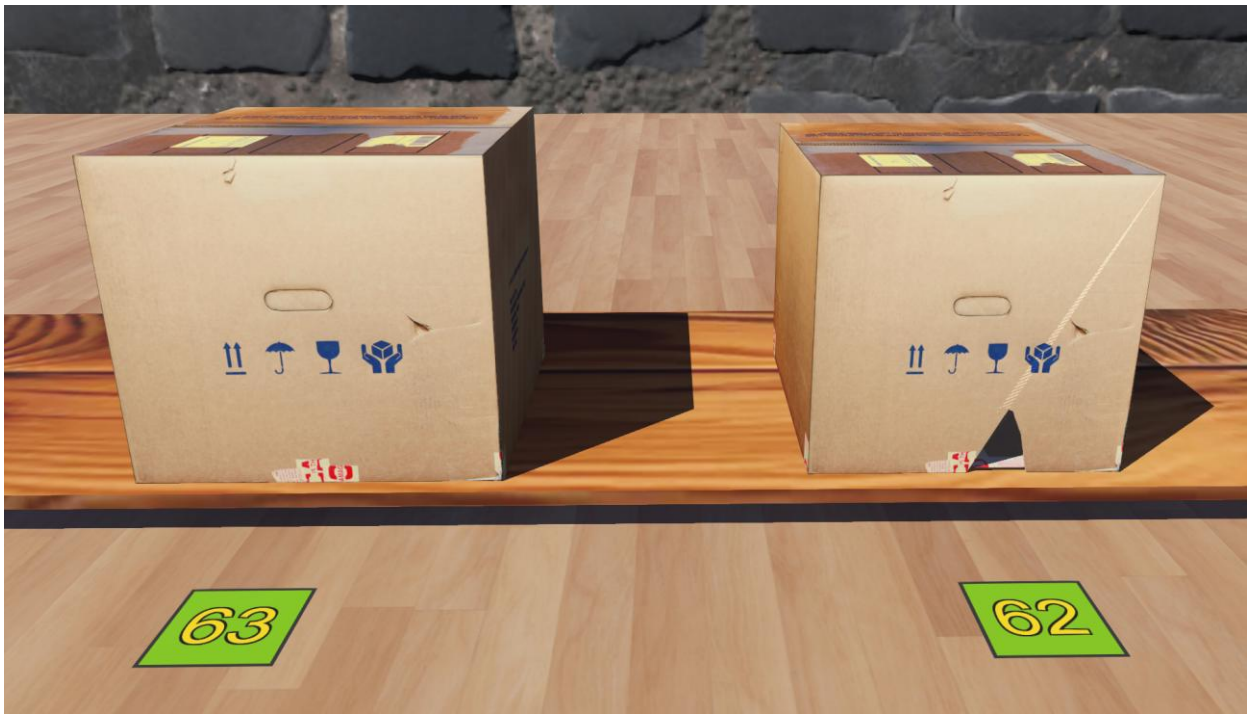

Figure 2.3 *- Comparison between an undamaged box (left) and a damaged box (right) showing procedurally generated holes and structural deformation. Both use the same base Cardboard texture.*

2.5 World Regeneration and Dataset Implications

Because the world is fully regenerated on each launch - with new random seeds governing damage type selection, damage geometry, empty position placement, and box orientations - no two simulation runs produce identical warehouses. This property was exploited during dataset collection: the simulation was run repeatedly in an automated loop, with the robot completing a full patrol and capturing images of all 64 box positions on each run. Over several hours of automated execution, this process accumulated the 16,797 images that constitute the training dataset for the classification model described in Section 5.

Randomization also means that every evaluation run tests the trained model against a warehouse it has never seen before, providing a form of built-in cross-validation that would be difficult and expensive to achieve with physical inspection data.

3. System Architecture

3.1 Design Philosophy

The central architectural decision in this project was to decouple the robot simulation from the desktop. Rather than building a monolithic application where a Webots controller, a classification model, and a user interface all run inside the same process, the system was designed as a set of independent services that communicate over the network. The result is a distributed architecture in which the simulation runs on a GPU workstation, the user interface runs in any web browser, and the AI services - image classification, speech recognition, language understanding, and speech synthesis - each run as isolated processes that can be deployed, scaled, or replaced independently.

This design has a practical origin: there is no robotics middleware that runs painlessly on all target platforms while simultaneously supporting a rich web-based interface. The two main candidates considered were ROS 2 and a custom Socket.IO-based protocol, and the trade-offs between them shaped the entire system.

3.2 Socket.IO vs ROS 2

ROS 2 is the industry standard for robotics middleware. It provides a mature publisher-subscriber messaging system, standardized message types (Twist, Pose, JointState), a rich ecosystem of navigation and manipulation packages (nav2, moveit, tf2), and well-documented integration with Webots on Linux. For a production multi-robot system, ROS 2 would be the expected choice.

However, several factors made it a poor fit for this project. First, compiling and running ROS 2 on Windows is significantly more complex than on Linux, introducing a heavy dependency chain and frequent build issues that would distract from the core project goals. Second, ROS 2's messaging system is designed for inter-process communication between C++ and Python nodes, not for streaming data to web browsers - bridging ROS topics to a browser requires additional infrastructure such as rosbridge or roslibjs, adding another layer of complexity. Third, the project involves a single robot in a controlled simulation with a fixed layout and known waypoints; the sophisticated localization, path planning, and transform management that ROS 2 excels at are largely unnecessary here.

Socket.IO, by contrast, provides a lightweight event-driven messaging layer with native browser support. It runs on any platform with Python or Node.js, requires no compilation, supports binary message transport for camera frames, and includes built-in room-based broadcasting. The key advantage for this project is that the same protocol that connects the robot controller to the server also connects the server to the browser, eliminating the need for a separate web bridge. The trade-off is the absence of robotics-specific primitives - there are no standard message types, no transform trees, no action servers - which means the project had to define its own command protocol. For a single-robot simulation with a known environment, this was a worthwhile exchange.

3.3 Component Overview

The system consists of five major components, each running as a separate process and communicating through a central server over Socket.IO and HTTP.

The Webots simulation runs the physics engine, renders the 3D environment, and executes the robot controller. The controller operates as a supervisor, meaning it has access to the simulation scene graph for reading ground truth data (box names, positions, damage labels) in addition to normal sensor access. Webots exposes its 3D view as a WebGL stream on a configurable port, which forms the visual foundation of the user interface.

The web server is the system's hub. Built with FastAPI and python-socketio, it launches and manages the Webots process, serves the client application, and routes all inter-component communication. Every command, status update, video frame, and inspection result pass through this server. It also manages the inventory state - tracking which boxes have been inspected, what the classifier predicted, and how those predictions compare against ground truth.

The web client is a single-page HTML/CSS/JavaScript application that renders the user interface as a transparent overlay on top of the Webots WebGL stream. The client connects to the server via Socket.IO for real-time events and uses HTTP for static resources and inventory queries. It provides control panels for robot commands, camera adjustments, voice interaction, and an inventory grid showing inspection progress and accuracy.

The box damage classification service runs the trained CNN model. It receives images over Socket.IO from the server, runs inference, and returns a classification (damaged, undamaged, or empty) with a confidence score. This service runs as an independent process, which means the model can be retrained and redeployed without modifying any other component.

The voice agent service provides speech-to-text, language understanding, and text-to-speech pipeline. It receives audio from the browser, transcribes it using Whisper, interprets the transcription using a local LLM, and returns both a structured command and a spoken response. This component is described in detail in Section 5.

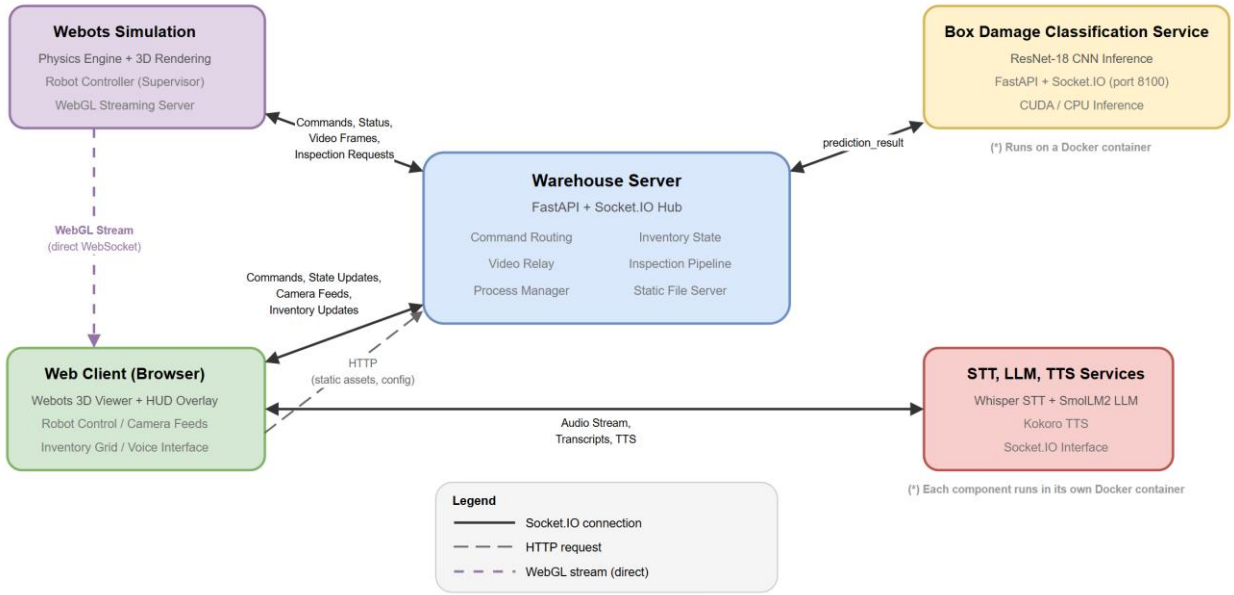

Figure 3.1 *- Distributed system architecture. All components communicate through the central FastAPI/Socket.IO server. Solid lines indicate Socket.IO connections; dashed lines indicate HTTP requests. The Webots WebGL stream connects directly to the browser.*

3.4 Communication Protocols

Command Protocol

All commands follow a uniform JSON structure with three fields: target (which subsystem should handle the command - robot, simulation, pedestrian, or c3), action (the specific operation), and params (a dictionary of action-specific parameters). The server's command handler dispatches incoming commands to the appropriate handler function based on the target field, executes the action, and emits a command_response event back to the requesting client. If the command modifies system state, a state_update event is also broadcast to all connected clients.

Robot commands include inspect (navigate to a specific box position), pause and resume (switch between standby and patrol modes), return (navigate to the start position), zoom (adjust camera field of view), and tilt (adjust camera pitch angle). Each command is relayed from the server to the robot controller's Socket.IO session.

Inspection Flow

The inspection pipeline is the system's most complex communication sequence and illustrates how the components collaborate. When the robot arrives at a box position and captures an image, it emits an inspection_request event containing the box identifier and the raw image bytes. The server saves the image to disk, forwards it to the box damage classification service, waits for the prediction result (with a 30-second timeout), updates the inventory state with the classification and its accuracy against ground truth, sends an inspection_response back to the robot so it can proceed to the next waypoint, and broadcasts an inventory_update to all connected UI clients so the inventory grid refreshes in real time.

Video Streaming

The robot controller captures frames from its cameras and emits them as binary video_frame events. The server uses Socket.IO rooms to efficiently relay these frames: clients that have joined the analytics room receive the frames for live video display, while clients that have not joined the room are not burdened with the bandwidth. This room-based approach means that adding video consumers (for example, a second monitoring display) requires no server-side changes.

Robot Registration and Status

When the robot controller starts, it emits a robot_register event. The server records its session identifier, which is subsequently used to route commands specifically to the robot rather than broadcasting them. The robot periodically emits robot_status events containing its current mode, position, heading, and the identifier of the box it is currently inspecting. The server relays these updates to all UI clients, which use them to update the status panel display.

3.5 Webots Web Viewer Integration

Webots includes a built-in Web Viewer that streams the simulation's 3D rendering as a WebGL scene to a <webots-view> custom HTML element. The project uses this capability to embed the live simulation view directly in the browser, but with an important customization: the native Webots toolbar and interaction controls are hidden via CSS, and a custom HTML/CSS overlay is rendered on top of the WebGL canvas.

The overlay is structured in two layers. The base layer is the <webots-view> element, which occupies the full viewport and renders the 3D warehouse scene. The UI layer floats above it with position: absolute and transparent backgrounds, containing the status panel, robot control buttons, chat interface, voice controls, and inventory grid. This approach gives the interface the appearance of a purpose-built 3D application with HUD-style controls, while the underlying rendering is handled entirely by Webots.

The WebGL stream connection is managed separately from the Socket.IO connection. On startup, the client requests the Webots streaming URL from the server via HTTP (/config endpoint), then initializes the <webots-view> element with that URL. If the connection drops, a reconnection overlay is displayed until the stream is re-established. Because the stream uses a standard WebSocket, it works through reverse proxies and across networks, enabling remote operation of the simulation from any browser with network access to the server.

3.6 Inventory State Management

The server maintains a persistent inventory state that tracks every box position in the warehouse. The state is initialized from the patrol_waypoints.json file generated alongside the world, which contains the ground truth condition (damaged, undamaged, or empty) for each position. As inspections occur, the state accumulates per-box records with the classification result, confidence score, timestamp, image path, and whether the prediction matched the ground truth.

Summary counters - total positions, inspected count, empty count, damaged count, undamaged count - are updated incrementally with each inspection and broadcast to all clients. An accuracy tracker maintains running counts of correct and incorrect predictions, along with a timestamped history that enables the UI to display how accuracy evolves as the patrol progresses. The entire state persisted to a JSON file after each update, so it survives server restarts.

When a new world is generated (which randomizes all box conditions and empty positions), the inventory state is reset, a new session identifier is generated, and a fresh captures directory is created to store the images from the new patrol run.

4. Robot Platform

4.1 Platform Selection

The project began with a Pioneer 3-DX, a differential-drive mobile robot widely available as a Webots PROTO. The Pioneer's two-wheel plus caster configuration is simple to control and well-understood, but it introduced a persistent problem during early testing: every heading change required the robot to rotate in place, and each rotation accumulated small odometry errors. Over the course of a full patrol - dozens of turns across four aisles - these errors compounded to the point where the robot's estimated position diverged noticeably from its actual position, causing it to miss waypoints and misalign with box positions.

The solution was to switch to the KUKA YouBot, an omnidirectional platform with four Mecanum wheels. Mecanum wheels have angled rollers that allow the robot to translate in any direction without changing its heading. This means the robot can move forward, sideways, or diagonally while always facing the same direction. By eliminating rotation from the navigation strategy entirely, the primary source of odometry drift was removed. The heading remains constant at zero throughout the patrol, and positional accuracy is maintained over long traversals.

4.2 Mecanum Wheel Kinematics

The YouBot's four Mecanum wheels are arranged in a rectangular configuration. The inverse kinematics map desired body velocities (vx, vy, omega) to individual wheel angular velocities using the robot's physical dimensions (wheel spacing and radius) and a standard four-wheel Mecanum model.

In practice, the controller sets omega to zero for all transit movements, using only vx and vy to reach each waypoint. This means the robot translates in any direction without ever rotating, keeping its heading fixed throughout the patrol. Rotation is reserved exclusively for fine heading corrections at waypoints, which occur rarely and with small magnitudes.

Forward odometry inverts this relationship. At each simulation timestep, the controller reads the four-wheel encoder deltas, computes the incremental body-frame displacements (dx, dy, d_theta), and integrates them into the global pose estimate.

Because omega is held at zero during movement, the angular term remains negligible and the odometry integrator operates in its most accurate regime (pure translation). This is the key advantage of the Mecanum drive for this application: by eliminating rotation, the primary source of odometry drift is removed, allowing reliable navigation across 64 waypoints with only a 5 cm tolerance and no external localization.

4.3 Custom PROTO and Sensor Configuration

The standard KUKA YouBot PROTO provides the base chassis, Mecanum wheels, and optionally a robotic arm. For this project, a custom YoubotDualCamera PROTO was created that removes the arm (unnecessary for visual inspection), adds a vertical camera mast, dual cameras with motorized tilt, obstacle detection sensors, SpotLights for shelf illumination, and a radio communication interface. The PROTO exposes all key parameters as configurable fields - mast height, camera resolution, field-of-view limits, tilt range, light

intensity, and beam angles - allowing the sensor configuration to be tuned without modifying the PROTO source.

The camera mast is a 0.9 m vertical cylinder mounted at the robot's center, topped by a housing that contains the two cameras. The mast elevates the cameras above the shelf surface to provide a downward viewing angle onto the boxes, which is important because the robot inspects boxes that sit on shelves at floor level while the robot travels alongside.

Two cameras (camera_left and camera_right) are mounted at the top of the mast, offset slightly along the X axis. The left camera faces backward (negative X direction) and the right camera faces forward (positive X direction). Since the robot always travels with its front toward positive X, the left camera observes boxes on the left side of the aisle (lower X, behind the robot) and the right camera observes boxes on the right side (higher X, ahead of the robot).

Each camera has 640 by 480 resolution, a default field of view of 1.0 radian, and a Zoom node that allows the field of view to be adjusted between 0.2 and 1.5 radians via the web interface. Also, each camera is mounted on a HingeJoint driven by a RotationalMotor (tilt_motor_left/tilt_motor_right) that provides motorized pitch adjustment between approximately -30 and +60 degrees, also controllable from the web interface. Position sensors on each tilt joint report the current angle. A SpotLight is co-located with each camera to illuminate the shelf area under inspection, with configurable intensity, beam width, and shadow casting.

Four infrared distance sensors (distance_front, distance_back, distance_left, distance_right) are mounted at the base of the robot body, each pointing outward along one of the cardinal directions. Each sensor uses a lookup table mapping 0 to 3 meters with an aperture of 0.3 radians and two rays. Before each movement step, the controller queries the sensor corresponding to the current movement direction and checks the reading against a threshold of 0.8 m. If an obstacle is detected, the robot stops and waits until the reading exceeds a clearance threshold of 1.0 m. The hysteresis between the stop and resume thresholds prevents oscillation when an obstacle is near the boundary. This mechanism allows pedestrians or other dynamic elements to cross the robot's path without causing navigation failures.

Wheel encoders on all four motors provide the incremental rotation measurements used for odometry. These are enabled at the simulation's basic timestep (32 ms) and read every control cycle. The robot has a total mass of 22 kg with an explicitly defined inertia matrix, and a center of mass offset to -4.5 cm on the Z axis to account for the low-mounted chassis.

Figure 4.1 *- The customized YouBot platform with vertical camera mast, dual tilt-motorized cameras, four infrared distance sensors, and Mecanum wheels. The robot maintains a fixed heading throughout patrol, using only lateral and longitudinal translation.*

4.4 Controller State Machine

The robot controller operates as a three-mode state machine: standby, patrol, and goto.

In standby mode, the robot is idle and waiting for commands. It processes incoming messages from the C3 channel and the Socket.IO connection but does not move. This is the initial mode on startup and the mode the robot returns to after completing a patrol or an individual command.

In patrol mode, the robot follows the pre-generated waypoint sequence autonomously. At each simulation step, it reads the next waypoint, navigates to it using proportional control on the body velocities, and checks whether the waypoint is a transit point (advance immediately) or an inspection point (stop and capture an image). When an inspection waypoint is reached, the controller captures an image from the appropriate camera, sends it to the server for classification via Socket.IO, waits for the response, and then advances to the next waypoint. When the final waypoint is reached, the controller prints a summary of waypoints visited and images captured, resets the waypoint index, and returns to standby.

In goto mode, the robot navigates to a specific position in response to a UI or voice command - typically to inspect a particular box. The controller uses aisle-aware path planning: if the target is in a different aisle, the robot first exits its current aisle by moving to the central corridor (Y = 0), then travels to the target aisle's X coordinate, and finally enters the aisle to reach the box. On arrival, it performs an inspection (if requested) and returns to standby.

UI commands received over Socket.IO have the highest priority and can interrupt any mode. A "pause" command immediately stops the robot and switches to standby. A "resume" command first navigates back to the patrol start position, then begins the patrol sequence. This is implemented as a transitional goto_then_patrol command that uses goto mode for the return trip and switches to patrol mode on arrival.

4.5 Navigation Strategy

The navigation approach is deliberately simple: pure odometry with no external position corrections and no rotation during transit. The robot moves toward each waypoint using proportional control, with the body-frame velocity vector pointing directly at the target. Speed is capped at 0.4 m/s to maintain control stability. The position tolerance is 5 cm - when the robot is within this distance of a waypoint, it is considered reached.

This strategy works because the environment is fully known and static (shelves and boxes do not move), the robot never rotates (eliminating the primary source of Mecanum odometry error), and the waypoint sequence is designed to keep the robot within aisles where deviations are bounded by the shelf walls. Over a complete patrol of all 64 box positions, the accumulated odometry error remains within acceptable bounds for image capture alignment.

The target heading is always zero. The calculate_target_theta method exists in the controller interface but always returns 0.0, reflecting the design decision that the robot's orientation is invariant. This is a direct consequence of the Mecanum drive choice: because the robot can translate in any direction, heading changes are unnecessary and avoiding them preserves odometry accuracy.

5. Methodology and Techniques

5.1 Dataset Collection

The classification model is trained entirely on synthetic images captured by the robot during automated patrol runs. The procedural world generation described in Section 2 produces a fresh warehouse layout on every launch, with randomized damage types, severity levels, box orientations, and empty positions. By running the simulation in an automated loop - regenerating the world, executing a full patrol, and collecting all 64 images per run - the system accumulated a dataset of 16,798 labeled images over several hours of execution on a workstation equipped with an NVIDIA RTX 4090 GPU.

Each image is a 640 by 480 pixel frame captured by one of the robot's two side-facing cameras as it pauses at an inspection waypoint. The ground truth label for each image - damaged, undamaged, or empty - is determined at world generation time and recorded in patrol_waypoints.json. Because the world is regenerated on every run, no two images share the same box geometry, damage pattern, or shelf arrangement, which provides the kind of visual diversity that prevents the model from memorizing specific textures or layouts.

The dataset exhibits a significant class imbalance. Undamaged boxes are the most common class, reflecting the default configuration where most boxes are intact. Damaged boxes are less frequent (governed by the damaged_box_ratio parameter), and empty positions are the least common (governed by the empty_ratio parameter, default 15%). The training split contains 13,435 images and the validation split contains 3,363 images (an 80/20 split).

5.2 CNN Architecture and Transfer Learning

The classification task maps each captured image to one of three classes: damaged, undamaged, or empty. The model uses ResNet-18 as the backbone, pretrained on ImageNet and fine-tuned on the collected dataset. ResNet-18 was selected for its favorable balance between accuracy and inference speed. Its 18-layer residual architecture is lightweight enough for real-time classification during patrol, where each image must be classified before the robot advances to the next waypoint, while its skip connections enable effective gradient flow during fine-tuning.

Transfer learning proceeds by loading the full ResNet-18 architecture with ImageNet-pretrained weights and replacing the final fully connected layer with a new linear layer mapping 512 features to three output classes. All layers are trainable - no feature extraction freezing is applied - allowing the pretrained convolutional filters to adapt to the specific visual characteristics of the synthetic warehouse images, which differ substantially from the natural photographic content of ImageNet.

The training script supports multiple backbone architectures (ResNet-18, ResNet-34, ResNet-50, and EfficientNet-B0) through a unified interface, allowing quick experimentation during development. The final model uses ResNet-18 based on empirical comparison across these architectures.

5.3 Training Configuration

Training uses cross-entropy loss with class-specific weights to compensate for the dataset imbalance. The weighting scheme computes inverse-frequency weights for each class -

classes with fewer training samples receive proportionally higher loss contributions. The resulting weight vector was approximately 1.21 for damaged, 2.05 for empty, and 0.59 for undamaged, reflecting the relative scarcity of each class in the training set.

Parameter	Value
Backbone	ResNet-18 (ImageNet pretrained)
Input size	224 x 224 pixels
Batch size	32
Optimizer	Adam (weight decay 1e-4)
Initial learning rate	0.001
LR scheduler	ReduceLROnPlateau (factor 0.5, patience 3)
Max epochs	50
Early stopping patience	10 epochs

Data augmentation is applied during training to improve generalization. The training pipeline applies random horizontal flips (50% probability), random vertical flips (30%), random rotation up to 15 degrees, and color jitter (brightness and contrast up to 20%, saturation up to 10%). Validation images receive only deterministic preprocessing - resize to 224 by 224 pixels and ImageNet-standard normalization - to ensure that validation metrics reflect the model's true performance on unaugmented captures.

5.4 Training Results

Training ran for 39 of the configured 50 epochs before early stopping triggered after 10 consecutive epochs without improvement in validation accuracy. The best model was saved at epoch 29, achieving a peak validation accuracy of 98.3%. Training loss decreased steadily from 0.267 in epoch 1 to 0.039 by epoch 39, while validation loss stabilized around 0.050 to 0.053 from epoch 20 onward, indicating that the model had converged without significant overfitting.

The confusion matrix on the 3,363-sample validation set reveals the model's per-class performance:

Class	Precision	Recall	Support
Damaged	99.9%	93.6%	923
Empty	100.0%	100.0%	548
Undamaged	97.0%	99.9%	1,891

The empty class is classified with perfect precision and recall - the visual difference between an empty shelf position and one occupied by a box is unambiguous. Undamaged boxes achieve near-perfect recall (99.9%) with high precision (97.0%), meaning that almost all undamaged boxes are correctly identified with only a small number of damaged boxes misclassified into this class. The damaged class achieves near-perfect precision (99.9%, meaning that when the model predicts damage it is almost always correct) and a recall of 93.6%, indicating that 59 out of 923 damaged boxes in the validation set were misclassified as undamaged.

These misclassifications typically correspond to boxes with light or subtle damage - small dents or minor tears - that produce visual differences too small for the model to distinguish from normal surface variation at the current rendering resolution. No damaged or undamaged boxes were ever confused with empty positions, and only a single undamaged

box was misclassified as damaged, representing a conservative error that is operationally acceptable (false alarms are preferable to missed damage in a quality inspection context).

5.5 Inference Service

The trained model is deployed as a standalone inference service that runs independently from the web server and the simulation. The service is built with FastAPI and Socket.IO, providing both REST endpoints for direct image submission and a Socket.IO interface for real-time integration with the warehouse server.

On startup, the service loads the saved checkpoint - which contains the model architecture name, class names, number of classes, and the trained state dictionary - and recreates the ResNet-18 architecture with the saved weights. The model runs on CUDA if available, falling back to CPU. The inference transform matches the validation preprocessing: resize to 224 by 224 pixels, convert to tensor, and normalize with ImageNet statistics.

For each prediction request, the service receives an image (as binary data over Socket.IO or as a file upload via REST), converts it to RGB, applies the transform, runs a forward pass with gradients disabled, and computes Softmax probabilities over the three classes. The result includes the predicted class, the confidence score, per-class probabilities, a `box_present` flag (false for the empty class), and an `is_damaged` flag (true only when the prediction is "damaged"). This structured output allows the warehouse server to update both the inventory display and the accuracy tracker in a single response.

The Socket.IO interface uses an asynchronous request-response pattern: the warehouse server emits a `predict` event with the image bytes and a `box_id`, and the inference service responds with a `prediction_result` event containing the classification result. The warehouse server tracks pending requests using a dictionary of `asyncio.Future` objects keyed by `box_id`, with a 30-second timeout per request. This design decouples the classification model from the rest of the system - the inference service can be retrained, updated, or redeployed without any changes to the warehouse server or the robot controller.

5.6 Voice Control Pipeline

The system supports voice-based robot interaction through a pipeline that chains three services: speech-to-text (STT), a language model (LLM), and text-to-speech (TTS). The pipeline runs end-to-end in the browser, with audio capture and playback handled client-side and the three processing stages running on the server.

Speech-to-Text

The browser captures audio using the MediaRecorder API and streams it to a Whisper-based STT service via the agent client SDK. The SDK manages a persistent Socket.IO connection to the voice agent server and provides event-based callbacks for interim and final transcription results. Interim results (`UserTranscript` events with `is_final: false`) update the UI in real time to provide visual feedback that speech is being recognized, while the final transcript triggers the next stage of the pipeline.

Language Model

The final transcript is sent to a local LLM for intent recognition and command generation. The LLM is a small, quantized model - SmolLM2-135M-Instruct in Q8 quantization - loaded

via llama.cpp with the LlamaCppEngine wrapper. This model was chosen for its minimal resource footprint: at 135 million parameters in 8-bit quantization, it runs efficiently on consumer hardware and responds with low latency, which is important for a conversational interface where the user expects near-immediate acknowledgment.

The LLM receives a carefully structured system prompt that defines the available robot operations and specifies the required JSON output format. Every response must include an Operation field (one of PATROL, STANDBY, INSPECT, RETURN, CHAT, ZOOM, or TILT) and a Response field containing natural-language text for speech synthesis. Some operations carry additional parameters - INSPECT requires a BoxId, ZOOM requires Action and Camera fields, and TILT requires an Action field. The system prompt includes few-shot examples for each operation type, establishing the mapping between natural language and structured commands.

The engine supports configurable generation parameters (temperature 0.2 for the robot agent to favor deterministic output, top-k 40, top-p 0.9, max tokens 256) and provides streaming output via generate_stream(), which yields content chunks through an async generator. A threading event allows the stream to be cancelled mid-generation if the user interrupts with new speech.

Conversation memory is maintained through a sliding-window strategy. The ThreadWindowMemory class keeps a per-thread message history and generates a context preamble by concatenating recent messages up to a character budget derived from the model's context window (8,192 tokens). The most recent messages are prioritized - the preamble is trimmed from the oldest end - so the model always has access to the immediate conversational context even as history grows.

Command Parsing

The LLM's JSON output is parsed client-side by the robotCommandParser. The parser extracts the JSON object from the response text (handling cases where the model produces extra text around the JSON), reads the Operation field, and dispatches accordingly. For PATROL, STANDBY, INSPECT, RETURN, ZOOM, and TILT operations, the parser invokes a command callback that emits the corresponding Socket.IO event to the warehouse server, which relays it to the robot controller. For the CHAT operation, no command is sent - the response text is simply spoken back to the user.

Text-to-Speech

The Response field from the LLM output is sent to a TTS service (Kokoro) for speech synthesis. The agent client SDK subscribes to TTSAudio events, which deliver synthesized audio as ArrayBuffer chunks. The browser plays these through the Web Audio API, providing the audible side of the conversational loop. The TTS voice and speed are configurable per session.

End-to-End Flow

A complete voice interaction proceeds as follows: the user speaks a command (for example, "inspect box 23"); the STT service transcribes the audio to text; the text is sent to the LLM, which produces a JSON response such as {"Operation": "INSPECT", "BoxId": "23", "Response": "Going to inspect box 23"}; the command parser extracts the INSPECT operation and emits a command event to the server with the box ID; the server relays the command to the robot controller, which navigates to box 23 and initiates an inspection; simultaneously, the

"Going to inspect box 23" text is synthesized and played back to the user as spoken confirmation.

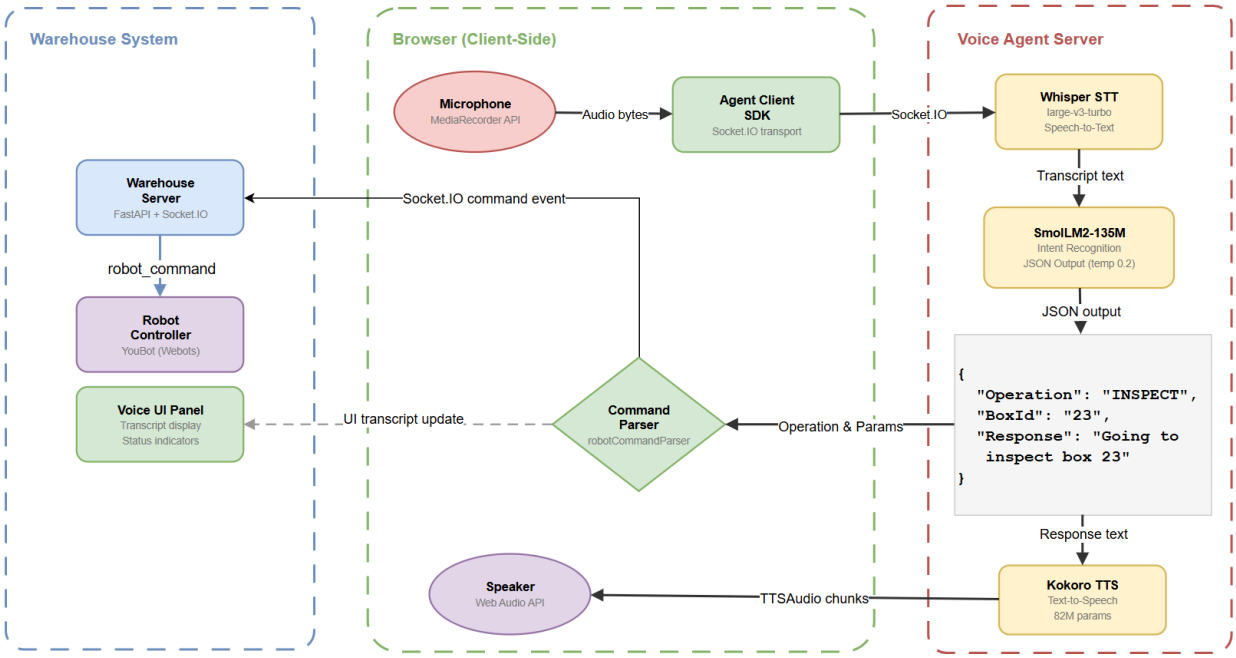

Figure 5.1 *- Voice control pipeline architecture. Audio flows from the browser microphone through STT transcription, LLM intent recognition with JSON-structured output, and TTS synthesis back to the speaker. Commands are dispatched in parallel to the robot controller via Socket.IO.*

5.7 Inventory Accuracy Tracking

The system provides real-time evaluation of the classification model's performance by comparing each prediction against the ground truth generated with the world. When the server receives a classification result from the inference service, it looks up the corresponding box's known condition in the ground truth data (loaded from patrol_waypoints.json) and records whether the prediction matches.

The accuracy tracker maintains running counters of correct and incorrect predictions, a cumulative accuracy percentage, and a timestamped history that records the accuracy after each inspection. This history is broadcast to all connected UI clients after every inspection, allowing the interface to display a live accuracy curve as the patrol progresses. If a box is re-inspected (for example, via a manual goto command), the tracker updates the counters to reflect the new prediction, adjusting for the previous result if it changed.

This built-in evaluation mechanism provides immediate feedback on model performance under varying conditions - different world configurations, different damage distributions, different empty ratios - without requiring a separate evaluation pipeline or post-hoc analysis.

6. Implementation

6.1 Project Structure

The project is organized into four independent services plus a shared configuration layer, each running in its own Docker container and communicating exclusively through Socket.IO and HTTP.

The **warehouse server** (server.py) is the central hub. It is a FastAPI application with an embedded Socket.IO server (via python-socketio with the ASGI transport) that serves the web interface, manages the Webots simulation process, routes commands between the UI and the robot controller, relays camera frames to connected browsers, proxies inspection images to the classification service, and maintains the inventory state. All static assets - the HTML interface, JavaScript modules, stylesheets, Webots Web Viewer libraries, and texture files - are served from this process. The server also launches and monitors the Webots simulation as a subprocess, passing the streaming port and world file as command-line arguments.

The **robot controller** (youbot_dual_camera.py) runs inside the Webots simulation process. It is a standard Webots controller written in Python that interfaces with the simulator's API for sensor readings and motor commands. It connects to the warehouse server as a Socket.IO client to receive UI commands and transmit camera frames and inspection requests. The controller loads its waypoint sequence from patrol_waypoints.json and its configuration from config.json.

The **classification service** (app.py) is a standalone FastAPI + Socket.IO application that loads the trained ResNet-18 model and serves predictions. It runs on port 8100 and communicates with the warehouse server through a dedicated Socket.IO connection. This separation allows the model to be retrained and redeployed without affecting the rest of the system.

The **voice agent server** is a separate service that hosts the STT, LLM, and TTS pipeline. It exposes a Socket.IO interface that the browser client connects to directly via the agentClient SDK. The LLM engine (llm_engine.py) wraps llama.cpp for local inference, and the thread memory manager (memory.py) maintains per-session conversation history.

The **world generator** (generate_world.py) is a build-time tool rather than a runtime service. It produces the .wbt world file and the patrol_waypoints.json ground truth file that the other components consume. The generator uses a modular damage system - dents.py, holes.py, tears.py - with shared geometry utilities (geometry.py) and a box management module (box.py) that handles placement, shelf assignment, and damage application.

Configuration is layered through two files: config.json for runtime parameters (server URLs, ports, feature flags, simulation settings) and agent_config.json for the voice pipeline configuration (STT model, LLM parameters, TTS voice). The config.py module provides a typed accessor for these settings.

6.2 Controller State Machine

The robot controller implements a three-mode state machine - standby, patrol, and goto - that runs within the Webots simulation loop. Each call to robot.step(timestep) executes one iteration of the control cycle: update odometry from wheel encoders, stream camera

frames to the server, process any pending UI commands, then execute the active mode's logic.

UI commands have the highest priority. The Socket.IO client runs in a background thread and appends incoming commands to a pending_ui_commands list. At the top of each control cycle, the main loop drains this list and calls handle_ui_command() for each entry. This handler can switch modes immediately - for example, receiving a mode:standby command clears the current command, empties the command queue, stops the wheels, and transitions to standby, regardless of what the robot was doing. A mode:patrol command does not switch directly to patrol mode; instead, it enters goto mode with a goto_then_patrol command that first navigates back to the patrol start position and then transitions to patrol on arrival.

In **standby** mode, the controller processes C3 (Command and Control Centre) messages from the Webots Emitter/Receiver radio channel and holds the robot stationary. The C3 channel provides an alternative command interface through a supervisor node that can issue inspect, goto, pause, and resume commands using a JSON message protocol.

In **patrol** mode, the controller iterates through the pre-loaded waypoint sequence. Each waypoint has a type - transit or inspection - and a movement direction. The inner state alternates between NAVIGATE (moving toward the current waypoint using proportional control) and INSPECT (stopping at an inspection waypoint to capture and classify an image). When the waypoint index reaches the end of the sequence, the controller prints a patrol summary, resets the index, and returns to standby. If the exit_on_complete flag is set (used during automated dataset collection), the controller terminates the simulation instead.

In **goto** mode, the controller executes a single navigation command - typically issued by the UI or voice interface to inspect a specific box. The execute_command() method implements aisle-aware path planning: if the target box is in a different aisle, the robot first exits its current aisle to the central corridor (Y = 0), translates to the target aisle's X coordinate, then enters the aisle to reach the box. On arrival, if the command includes an inspection, the controller captures an image and sends it to the server. When the command completes, the mode reverts to standby.

The inspect_box() method handles the classification handshake with the server. It selects the appropriate camera based on the box's side of the aisle, captures a JPEG image, sends it as binary data in an inspection_request Socket.IO event along with the box ID, and then blocks - polling robot.step() in a loop - until the server responds with an inspection_response event or a 30-second timeout expires. An asyncio.Event-like mechanism (using a simple flag and result variable set from the Socket.IO callback thread) synchronizes the response across threads without using timeouts or sleep-based polling.

6.3 Server Routing

The warehouse server acts as a message router between three categories of clients: browser UIs, the robot controller, and the classification service. The server tracks the robot's Socket.IO session ID (robot_sid) at registration time and uses it for targeted message delivery.

The primary routing path for commands follows a uniform protocol. The browser emits a command event containing a target (robot, pedestrian, c3, or simulation), an action (inspect, patrol, standby, return, zoom, tilt, etc.), and optional parameters. The server's command handler dispatches to the appropriate handler function based on the target field. For robot

commands, handle_robot_command() translates the UI action into the controller's command vocabulary and emits a robot_command event to the robot's session. For example, a UI "inspect" action with box_id: "23" is relayed as a robot_command with action: "goto" and params: {position_id: "23"}. The server sends a command_response back to the originating client (success or error) and broadcasts a state_update to all connected clients so that multiple browser windows stay synchronized.

The inspection flow is the most complex routing path. When the robot controller reaches an inspection waypoint and captures an image, it emits an inspection_request event with the image bytes and box ID. The server's handle_inspection_request handler executes a four-step pipeline: save the image to the session's captures directory; forward the image to the classification service via call_box_damage_service() (which emits a predict event on the dedicated box-damage Socket.IO connection and waits for a prediction_result response using an asyncio.Future with a 30-second timeout); update the inventory state with the classification result; and finally emit an inspection_response to the robot (so it can advance to the next waypoint) and broadcast an inventory_update to all browser clients (so the UI refreshes). If the classification service is unavailable, the server still records the inspection with an "unknown" status and sets a service_error flag in the broadcast so the UI can display a warning.

Camera streaming follows a simpler path. The robot controller emits video_frame events containing JPEG-compressed image bytes and a camera identifier. The server rebroadcasts these as process_video events to all clients that have joined the analytics room (which happens automatically on connection). This uses Socket.IO rooms rather than individual emits to avoid per-client overhead.

6.4 Frontend Interface

The web interface is structured as two visual layers rendered in the browser. The base layer contains the Webots 3D viewer - a <webots-view> custom element that connects to the Webots streaming server via WebSocket and renders the simulation scene using WebGL. The overlay layer contains a set of draggable, resizable HUD panels that float above the 3D scene, providing robot control, status monitoring, and data visualization without leaving the simulation view.

Panel System

The MenuManager class manages the panel lifecycle. It discovers all HUD panel elements by their CSS class, creates a bottom menu bar with toggle buttons for each panel (using SVG icons), and attaches drag and resize handlers via the Moveable library. Panels can be shown, hidden, repositioned, and resized independently. A z-index manager ensures that clicking a panel brings it to the front. Non-resizable panels (status, simulation controls, voice) are locked to their default dimensions. The menu bar itself is fixed at the bottom center of the viewport.

The interface provides nine panels: Status (server and Webots connection indicators), Robot Control (mode toggle, return button, inspect-by-ID input), Pedestrian Control (for the simulated warehouse worker), Simulation (playback controls and elapsed time), Voice Assistant (microphone button and conversation transcript), Browser Console (redirected console.log output), Webots Console (simulation stdout/stderr), Robot Cameras (live front and rear camera feeds), and Inventory State (classification results and accuracy tracking).

Camera Feed Display

The camera panel displays two live feeds from the robot's left and right cameras. The server broadcasts process_video events containing JPEG binary data and a camera identifier. The client creates a Blob URL from the binary data and sets it as the src of the corresponding <img> element. Previous Blob URLs are explicitly revoked on each frame to prevent memory leaks. The feeds update at the rate the controller streams frames, which is tied to the simulation timestep.

Simulation Controls

The simulation panel provides playback controls (play, pause, step, reset, fast-forward) and a time display. These controls interact with the Webots Web Viewer's internal WebSocket connection. The hookWebotsWebSocket() function intercepts the viewer's WebSocket messages to extract simulation time updates, which are formatted and displayed in the time counter. Playback state is synchronized across all connected clients through sim_state_changed events - when one browser pauses the simulation, all others update their UI to reflect the change.

Keyboard Shortcuts

The interface binds keyboard shortcuts for common operations: K for play/pause, period for step, R for reset, F for fast mode, number keys 1 through 9 to toggle specific panels, and 0 to hide all panels. These are registered in initKeyboardShortcuts() and are disabled when an input field has focus.

6.5 Inventory Tracking

The inventory system spans the server and the frontend, providing a real-time visual comparison between the CNN's classification output and the ground truth labels generated with the world.

Server-Side State

The InventoryState class on the server manages a persistent JSON store (inventory_state.json) that tracks the status of all 64 box positions. On initialization, it loads the ground truth from patrol_waypoints.json - the same file the world generator produces and the robot controller uses for waypoint navigation. Each ground truth entry records whether the position is empty, occupied with an undamaged box, or occupied with a damaged box.

When update_box() is called with a classification result, the method records the inference outcome (classification label, confidence score, image path, timestamp), looks up the corresponding ground truth entry, and determines whether the prediction is correct. It maintains running counters - correct, incorrect, and total evaluated - and appends a timestamped entry to the accuracy history. The summary statistics (counts by class: inspected, empty, damaged, undamaged) are recomputed on each update. If a box is re-inspected, the method adjusts the counters to account for the previous prediction before applying the new one, so the accuracy figure always reflects the most recent classification for each box.

The full inventory state is sent to new clients on connection (inventory_full event) and individual updates are broadcast after each inspection (inventory_update event). The state is also persisted to disk so it survives server restarts within the same session.

Frontend Grid Display

The inventory panel renders an 8 by 8 grid that mirrors the physical warehouse layout. The initInventoryGrid() function creates 64 box cells organized into two halves (left and right) separated by a gap that represents the central corridor, with aisle separators after every two rows to match the four-aisle warehouse geometry. Each cell displays the box number, a color-coded background indicating the inference result (grey for uninspected, blue for empty, green for undamaged, red for damaged), and a thin bottom bar showing the ground truth label using the same color scheme. When the inference matches the ground truth, the cell receives a match class; when they differ, it receives a mismatch class with a distinct visual indicator. Clicking any cell opens a modal that displays the captured image alongside the inference and ground truth labels.

Above the grid, the panel shows the current accuracy percentage, a correct/total counter, class-wise summary statistics, and a progress bar indicating how many of the 64 positions have been inspected. All of these updates in real time as inventory_update events arrive during a patrol.

6.6 Robot Command Parser

The RobotCommandParser is a client-side module that bridges the voice control pipeline and the robot control interface. It receives the raw text output from the LLM - which may contain a JSON object wrapped in extra text - and converts it into concrete robot commands.

The parsing logic first attempts a direct JSON.parse() on the trimmed text. If that fails (because the LLM produced explanatory text around the JSON), it extracts the first JSON-like substring using a regex match on curly braces and parses that. If no valid JSON with an Operation field is found, the text is treated as plain conversational output and forwarded to the chat handler for TTS playback.

For valid commands, the parser dispatches on the Operation field. PATROL, STANDBY, and RETURN map directly to mode changes. INSPECT extracts a BoxId parameter and includes it in the command payload. ZOOM and TILT extract action and camera parameters with validation against allowed values (in/out/reset for zoom; up/down/reset for tilt). CHAT produces no robot command - only the response text is forwarded for speech synthesis. The parser also recognizes and silently discards operations that are input-only markers (STATUS_INFO, INSPECTION_RESULT, OBSTACLE_DETECTED, OBSTACLE_CLEARED) - these are event types that the system sends to the LLM for context but that the LLM should never output as commands.

The parser uses a callback-based architecture: the constructor accepts onCommand, onChat, onError, and onRawResponse handlers. In the main application, the onCommand callback emits the appropriate Socket.IO event to the warehouse server (using the same command protocol as the UI buttons), and the onChat callback routes the response text to the TTS pipeline. A static getConfirmation() method provides human-readable descriptions of each command type, used for generating fallback TTS responses when the LLM does not include a spoken response in its output.

7. Results and Evaluation

7.1 Classification Performance

The ResNet-18 model was trained for 39 epochs before early stopping halted training after 10 consecutive epochs without improvement in validation accuracy. The best checkpoint was saved at epoch 29, achieving a validation accuracy of 98.3%. Figure 7.1 shows the training dynamics.

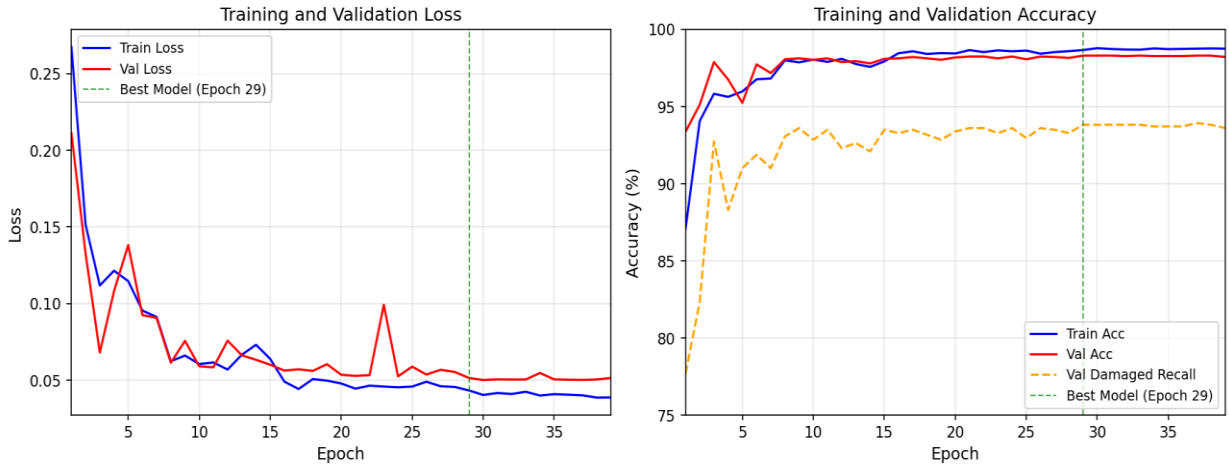

Figure 7.1 *- Training curves. Left: training and validation loss. Right: overall accuracy and damaged-class recall. The green dashed line marks epoch 29, where the best model was saved. Validation loss stabilizes around 0.050 from epoch 20 onward, while training loss continues decreasing to 0.039.*

The loss curves show rapid initial convergence, with both training and validation loss dropping sharply in the first three epochs. By epoch 10, both curves are below 0.07 and the model has already surpassed 98% validation accuracy. The remaining epochs produce only marginal improvements, with the best checkpoint at epoch 29 representing a 0.3 percentage point gain over epoch 10. The train-validation gap remains small throughout, confirming that the data augmentation and class weighting provide sufficient regularization to prevent overfitting.

The accuracy plot reveals that damaged-class recall stabilizes around 93 to 94% from approximately epoch 15 onward, consistently below the overall accuracy. This gap reflects the inherent difficulty of distinguishing lightly damaged boxes from undamaged ones - a structural limitation of the visual signal rather than a training insufficiency.

Per-Class Analysis

The confusion matrix on the 3,363-sample validation set (Figure 7.2) breaks down the model's error patterns.

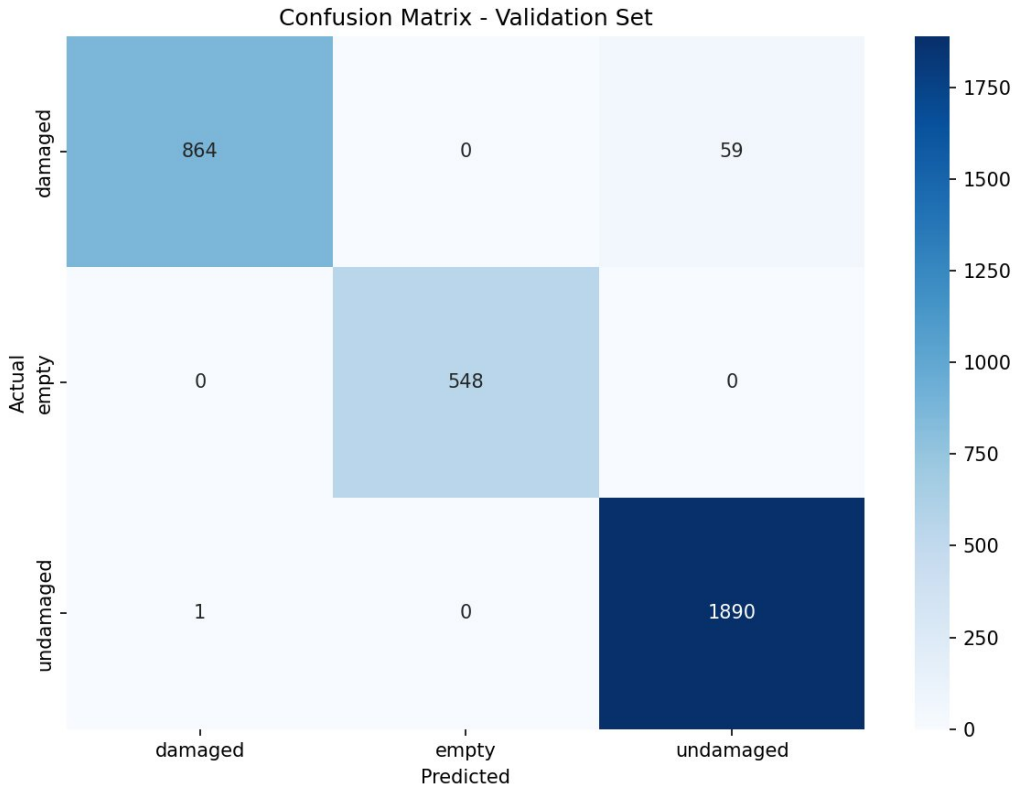

Figure 7.2 *- Confusion matrix on the validation set. The empty class is classified perfectly. The primary error mode is damaged boxes misclassified as undamaged (59 out of 923).*

Class	Precision	Recall	F1 Score	Support
Damaged	99.9%	93.6%	96.6%	923
Empty	100.0%	100.0%	100.0%	548
Undamaged	97.0%	99.9%	98.4%	1,891
Overall				**3,363**

Three observations emerge from the confusion matrix. First, the empty class is perfectly separable - no empty shelf position was ever confused with a box, and no box was confused with an empty position. This is expected since the visual difference between an empty shelf and a cardboard box is unambiguous regardless of viewing angle or lighting. Second, the damaged-to-undamaged confusion is overwhelmingly one-directional: 59 damaged boxes were misclassified as undamaged, while only a single undamaged box was misclassified as damaged. This asymmetry means the model is conservative about predicting damage - it requires strong visual evidence to flag a box - which results in near-perfect precision (99.9%) for the damaged class but lower recall (93.6%). Third, there is zero confusion between either box class and the empty class, meaning the model's feature representations cleanly separate "box present" from "no box" as a first-order distinction, with damage detection operating as a finer-grained second-order classification.

The 59 false negatives (damaged boxes classified as undamaged) are concentrated among boxes with subtle damage - small dents or minor surface tears that produce minimal visual deviation from an intact surface at the 640 by 480 capture resolution and the typical camera-to-box distance during inspection. The single false positive (one undamaged box classified as damaged) confirms that the model's damage predictions are highly reliable - when the model flags damage, it is correct in all but the rarest cases.

7.2 Navigation Performance

The robot uses pure odometry for navigation (as described in Section 4.5), with no external localization corrections. This approach is viable because the Mecanum wheels operating on a flat warehouse floor introduce minimal systematic drift, and the robot's omega is constrained to zero during transit segments to prevent heading errors from accumulating.

Empirically, the odometry-based navigation achieves consistent waypoint arrival within the 5 cm tolerance threshold across full patrol runs of 64 inspection stops. The proportional speed controller brings the robot smoothly to each target position, and the aisle-aware path planning ensures that the robot never attempts diagonal movements that would cause unpredictable Mecanum wheel interactions with shelf edges.

The waypoint sequence is designed so that every aisle transit starts from the central corridor (Y = 0), which acts as a natural reference line. Even if small odometry errors accumulate during an aisle traversal, the return to the corridor resets the lateral position error. The `aisle_x` field in each waypoint provides an additional correction anchor - during forward/backward movement within an aisle, the controller applies proportional correction toward the nominal aisle center X coordinate, preventing gradual lateral drift away from the shelf face.

The IR distance sensors (3 m range, 0.8 m activation threshold) provide a safety layer rather than a navigation input. When an obstacle is detected in the direction of travel, the controller stops and emits an `obstacle_detected` event. The obstacle does not cause rerouting - in the structured warehouse environment, unexpected obstacles are rare - but the event is forwarded to the LLM via the voice pipeline so that the system can verbally report the situation to the operator.

7.3 System Integration

Voice Command Pipeline

The end-to-end voice pipeline - from spoken command to robot action - operates with a latency of approximately 2 to 4 seconds depending on utterance length. The Whisper STT service contributes the largest portion of this latency; the SmolLM2-135M LLM responds in under 500 ms for typical command inputs due to its small parameter count and quantized inference. TTS synthesis runs in parallel with command dispatch, so the user hears spoken confirmation while the robot is already moving.

The LLM's structured JSON output format, enforced through the system prompt with few-shot examples and a low temperature of 0.2, produces valid parseable commands in the large majority of interactions. When the LLM output does not parse as JSON - typically because of an ambiguous or off-topic utterance - the `RobotCommandParser` falls back to treating the output as conversational text and routes it to TTS, so the user receives a spoken response even when no robot command is executed.

The command vocabulary covers the primary robot operations: starting and stopping patrol, inspecting specific boxes by ID, returning to the start position, and adjusting camera zoom and tilt. The CHAT operation handles general conversation that does not map to any robot action, allowing the system to respond naturally to questions about its status or the warehouse environment.

Real-Time Inventory

The inventory tracking system provides immediate visual feedback during patrol. As the robot inspects each box, the classification result appears on the 8 by 8 grid within the time it takes for the image to be captured, transmitted to the classification service, classified, and broadcast back to the UI - typically under 2 seconds from the moment the robot stops at an inspection waypoint. The ground truth overlay allows the operator to spot misclassifications in real time by observing cells where the inference color does not match the ground truth bar.

The accuracy counter updates after every inspection, providing a running evaluation that converges toward the model's true performance as more boxes are classified. For a typical patrol with the default damage ratio and empty ratio, the in-simulation accuracy aligns closely with the offline validation accuracy reported in Section 7.1, confirming that the model generalizes from the training data to novel world configurations.

7.4 Damage Visibility Analysis

The model's 93.6% recall on damaged boxes - a substantial improvement over earlier iterations but still the weakest per-class metric - prompted an investigation into which damage types are most difficult to detect. The procedural damage system generates three categories - dents, holes, and tears - with randomized severity, position, and orientation. Visual inspection of the misclassified samples revealed that the detection difficulty correlates strongly with damage severity and type.

Holes and tears, which produce clearly visible openings or flaps on the box surface, are detected with high reliability even at small sizes. Their visual signatures - sharp edges, shadows inside openings, displaced material - are distinctive features that the CNN learns effectively.

Dents are the most challenging category. A shallow dent on a cardboard surface produces only a subtle change in the normal surface, which manifests as a slight shadow or highlight shift in the rendered image. At the 640 by 480 capture resolution and the typical camera-to-box distance (approximately 0.3 to 0.5 meters), minor dents occupy only a few pixels of altered shading. The model's feature extraction at the 224 by 224 input resolution further down samples this signal. The inverse-frequency class weighting biases the model toward detecting damage, but it cannot fully compensate for cases where the visual evidence is genuinely ambiguous at the available resolution.

Several factors could improve damaged class recall in future iterations. Increasing the camera resolution from 640 by 480 to 1280 by 960 would provide four times the pixel density on the damage region. Reducing the camera-to-box distance (by adjusting the inspection waypoint positions or using the zoom motor to magnify the view) would increase the apparent size of damage features. Adjusting the procedural generation to set a minimum damage severity threshold would eliminate training examples where the damage is visually imperceptible, sharpening the model's decision boundary. Alternatively, a multi-scale inspection strategy - first classifying at the current distance, then zooming in for a second classification if the first result is uncertain - could improve recall without requiring changes to the training data.

8. Conclusions and Future Work

8.1 Summary

This report presented the design and implementation of an autonomous warehouse inspection robot built entirely within the Webots simulation environment. The system integrates a KUKA YouBot equipped with dual cameras and Mecanum drive, a CNN-based damage classification service, a voice control pipeline, and a real-time web interface - all coordinated through a distributed Socket.IO architecture.

The core inspection task - autonomously navigating a four-aisle warehouse and classifying the condition of 64 box positions - is accomplished with a validation accuracy of 98.3%. The ResNet-18 model, trained exclusively on synthetic images generated by the simulation itself, achieves perfect classification of empty positions, 99.9% recall on undamaged boxes, and 93.6% recall on damaged boxes. The primary limitation is the detection of subtle surface damage (shallow dents) at the current capture resolution and camera distance.

The navigation system uses pure odometry with Mecanum kinematics and zero-omega transit, achieving reliable waypoint arrival within a 5 cm tolerance across full patrol runs without external localization. The aisle-aware path planner handles transitions between the central corridor and inspection aisles, and infrared sensors provide obstacle detection as a safety layer.

The voice interface allows an operator to control the robot through natural speech, with a pipeline that chains Whisper STT, a SmolLM2-135M LLM for intent recognition, and Kokoro TTS for spoken feedback. The LLM produces structured JSON commands that the client-side parser dispatches to the robot controller, supporting patrol control, targeted inspections, camera adjustments, and general conversation.

The web interface overlays draggable HUD panels on the Webots 3D viewer, providing live camera feeds, robot control, simulation playback, console output, and an inventory grid that visualizes classification results against ground truth in real time. The inventory tracker computes running accuracy as the patrol progresses, giving immediate feedback on model performance under each world configuration.

8.2 Key Takeaways

The project required several developments that went beyond assembling individual components.

The **procedural world generation** pipeline produces a fully randomized warehouse on every launch - box placements, damage types and severities, empty positions, and shelf arrangements are all regenerated - which enabled the collection of 16,797 diverse training images without manual annotation. The ground truth labels are embedded in the world definition and flow automatically to both the training pipeline and the runtime accuracy tracker, eliminating the annotation bottleneck that typically gates supervised learning on synthetic data.

The **distributed Socket.IO architecture** replaces the ROS2 middleware that is conventional in robotics with a lightweight web-native protocol. This decision traded the ecosystem of ROS2 packages for a simpler integration path with the Webots Web Viewer and browser-based interfaces, and it enabled the entire system - simulation, server,

classification service, and voice agent - to run in Docker containers with standard web networking rather than requiring ROS2's DDS discovery and message serialization infrastructure.

The **custom YouBot PROTO** with dual side-facing cameras and motorized zoom and tilt addresses the specific geometry of a warehouse aisle inspection, where boxes are on both sides and the robot must capture images without rotating. The Mecanum drive and zero-omega navigation strategy complement this by allowing the robot to traverse aisles in a straight line while both cameras face the shelves.

The **integrated evaluation loop** - where the same ground truth that defines the simulation world is used to score the model's predictions in real time - provides a closed-loop development workflow. Changes to the damage generation, camera configuration, or model architecture can be evaluated immediately by running a patrol, without a separate offline evaluation step.

8.3 Limitations

Several limitations constrain the current system. The navigation relies on a pre-computed waypoint sequence tied to a known warehouse layout. The robot cannot adapt to layout changes, discover new aisles, or plan paths in an unknown environment. The odometry-only localization, while sufficient on the flat simulated floor, would accumulate drift on real-world surfaces with wheel slip.

The classification model operates on single frames captured at fixed inspection points. It does not use temporal information (multiple views of the same box from different angles) or depth data, both of which could improve detection of ambiguous damage. The 93.6% damaged-class recall, while strong for an automated first-pass inspection, would benefit from further improvement for safety-critical applications.

The voice pipeline depends on a small LLM (135M parameters) that handles the defined command vocabulary reliably but has limited capacity for complex multi-turn reasoning or disambiguation of ambiguous commands. The system prompt and few-shot examples compensate for the model's limited capacity, but edge cases - such as commands that reference previous context or require spatial reasoning - may produce incorrect parses.

8.4 Future Work

Several extensions would strengthen the system's capabilities and move it closer to real-world applicability.

Object detection for unknown layouts. Replacing the fixed waypoint sequence with a YOLO-based detection pipeline would allow the robot to identify box positions dynamically, enabling operation in warehouses where the layout is not known in advance or changes over time. The detector could locate boxes on shelves from the camera stream during a preliminary exploration pass, after which the inspection planner would generate waypoints on-the-fly.

Multi-robot coordination. The Socket.IO architecture already supports multiple simultaneous robot connections. Extending the server with a task allocation layer - distributing aisles or box subsets across multiple robots - would enable parallel inspection

of large warehouses. The inventory state is already centralized on the server, so multiple robots can contribute classifications to the same session without conflicts.

Serpentine patrol optimization. The current patrol visits all boxes in a fixed sequence that traverses each aisle end-to-end. A serpentine pattern - alternating the direction of travel in successive aisles - would reduce the number of corridor transits and shorten the total patrol time. The waypoint generator could be extended to produce both linear and serpentine sequences based on a configuration flag.

Multi-scale inspection. A two-pass strategy where uncertain classifications trigger a closer re-inspection - using the zoom motor to magnify the region of interest - could improve damaged-class recall without retraining the model. The confidence score returned by the inference service provides a natural threshold for deciding when a second look is warranted.

Sim-to-real transfer. The procedural damage generation and synthetic training pipeline are designed with domain randomization principles in mind. Validating the trained model on real cardboard box images - and quantifying the sim-to-real accuracy gap - would establish the practical value of the synthetic training approach and guide the addition of domain adaptation techniques (style transfer, texture randomization, lighting variation) to close any gap.

9. References

[1] Cyberbotics Ltd., "Webots: Open-Source Robot Simulator"
https://cyberbotics.com/.
Webots is an open-source robot simulation software providing a complete development environment for modeling, programming, and simulating robots. It was used as the simulation platform for the warehouse environment and robot controller.

[2] Cyberbotics Ltd., "Webots Web Viewer"
https://deepwiki.com/cyberbotics/webots/5.1-web-viewer.
The Webots Web Viewer is a browser-based 3D viewer component that enables users to view and interact with Webots simulations through a web interface, providing remote visualization and control without requiring the full Webots desktop application.

[3] KUKA Roboter GmbH, "KUKA youBot"
https://cyberbotics.com/doc/guide/youbot
The KUKA youBot is an omnidirectional mobile platform with a 5-DOF manipulator arm, used in the project as the base robot platform with a custom dual-camera PROTO extension.

[4] PyTorch
https://pytorch.org/
PyTorch is an open-source deep learning library, originally developed by Meta Platforms and currently developed with support from the Linux Foundation. It was used for model training, transfer learning, and inference.

[5] M. Tan and Q. V. Le, "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks," in *Proc. ICML*, 2019, pp. 6105–6114.
https://arxiv.org/abs/1905.11946.
EfficientNet introduced a compound scaling method for convolutional neural networks. EfficientNet-B0 was evaluated as a candidate backbone during model development.

[6] A. Radford et al., "Robust Speech Recognition via Large-Scale Weak Supervision," in *Proc. ICML*, 2023.
https://arxiv.org/pdf/2212.04356
Whisper is a state-of-the-art model for automatic speech recognition and speech translation, trained on 680,000 hours of multilingual data. The Whisper large-v3-turbo variant, a pruned version with 4 decoding layers instead of 32, was used as the STT component for its balance of speed and accuracy. Model available at: https://huggingface.co/openai/whisper-large-v3-turbo.

[7] L. Ben Allal et al., "SmolLM2: When Smol Goes Big - Data-Centric Training of a Small Language Model," 2025.
https://arxiv.org/pdf/2502.02737
SmolLM2 is a family of compact language models available in 135M, 360M, and 1.7B parameter sizes, capable of solving a wide range of tasks while being lightweight enough to run on-device. The 135M-Instruct variant was used for voice command intent recognition. Model available at: https://huggingface.co/HuggingFaceTB/SmolLM2-135M-Instruct.

[8] Hexgrad, "Kokoro: Lightweight Text-to-Speech Model"
https://github.com/hexgrad/kokoro.
Kokoro is an open-weight TTS model with 82 million parameters that delivers quality comparable to larger models while being significantly faster and more cost-efficient. Released under the Apache license, it was used as the speech synthesis component of the voice pipeline.

[9] G. Georgiou, "llama.cpp: LLM Inference in C/C++"
https://github.com/ggerganov/llama.cpp
A high-performance C/C++ implementation for running large language model inference, used as the backend for the SmolLM2 LLM engine via the Python bindings.

[10] Socket.IO
https://socket.io/
Socket.IO is a library that enables low-latency, bidirectional, and event-based communication between a client and a server. It was used as the primary communication protocol across all system components.

[11] S. Ramirez, "FastAPI"
https://fastapi.tiangolo.com/
FastAPI is a modern, high-performance web framework for building APIs with Python based on standard Python type hints. It was used as the HTTP framework for both the warehouse server and the classification inference service.

[12] Encode, "Uvicorn: ASGI Web Server"
https://uvicorn.dev/
Uvicorn is an ASGI web server implementation for Python, used to serve the FastAPI applications with async support.

[13] python-socketio
https://python-socketio.readthedocs.io/
Python implementation of the Socket.IO protocol, used on the server side with the ASGI transport for integration with FastAPI.

[14] Docker Inc., "Docker: Accelerated Container Application Development"
https://www.docker.com/
Docker was used to containerize all system services, providing isolated and reproducible deployment of the warehouse server, classification service, voice agent, and simulation environment.

[15] Daybrush (Younkue Choi), "Moveable.js"
https://github.com/daybrush/moveable
Moveable is a JavaScript library providing draggable, resizable, scalable, and rotatable functionality for DOM elements. It was used to implement the draggable and resizable HUD panel system in the web interface.

[16] nginx: HTTP and Reverse Proxy Server"
https://nginx.org/
Used as a reverse proxy for routing HTTP and WebSocket traffic between the browser and the backend services within the Docker network.